Delivering Personalized Movie Recommendations with an AI-Driven Matchmaking System

Student Name: [Dharshini B]

**Register Number:** [621823104009]

**Institution:** [PGP College of Engineering and Technology]

**Department:** [CSE]

**Date of Submission:** [19/05/2025]

**Github Repository Link:** []



# Problem Statement

*Problem Existing: movie recommendation systems often rely on simplistic approaches, such as genre-based or rating-based recommendations, which fail to capture the nuances of individual user preferences. This leads to:*

* 1. *Sub optimal Recommendations: Users receive suggestions that don't align with their tastes.*
  2. *Lack of Discovery: Users miss out on hidden gems and new releases that might interest them.*
  3. *User Fatigue: Users become disengaged due to repetitive or irrelevant recommendations.*

***Goal*:** *Develop an AI-driven matchmaking system that delivers personalized movie recommendations, enhancing user experience and engagement.*

# Abstract

*This project aims to develop an AI-driven matchmaking system for delivering personalized movie recommendations. The existing recommendation systems often fail to capture individual user preferences, leading to sub optimal suggestions. Our*

*objective is to create a system that accurately matches users with movies they'll enjoy. We approach this problem by leveraging machine learning algorithms and natural language processing techniques to analyze user behavior and movie features. Our system generates personalized recommendations, enhancing user experience and engagement*

# System Requirements

Hardware requirements:

* 1. RAM: 8 GB (minimum), 16 GB (recommended)

* 1. Processor: Intel Core i5 or equivalent (minimum), Intel Core i7 or equivalent (recommended)

* 1. GPU: Optional, but recommended for faster computation (e.g., NVIDIA GeForce)

software requirements:

1. Python Version: Python 3.8 or later

1. Required Libraries:
   * TensorFlow or PyTorch for deep learning

* + Scikit-learn for machine learning

1. IDE:
   * Jupyter Notebook or Jupyter Lab
   * Google Colab (optional)

1. Additional Tools:
   * Data storage: relational databases (e.g., MySQL) or NoSQL databases (e.g., MongoDB)

# Objectives

Primary objective: Develop an AI-driven movie recommendation system that provides personalized suggestions to users, enhancing their experience and engagement.

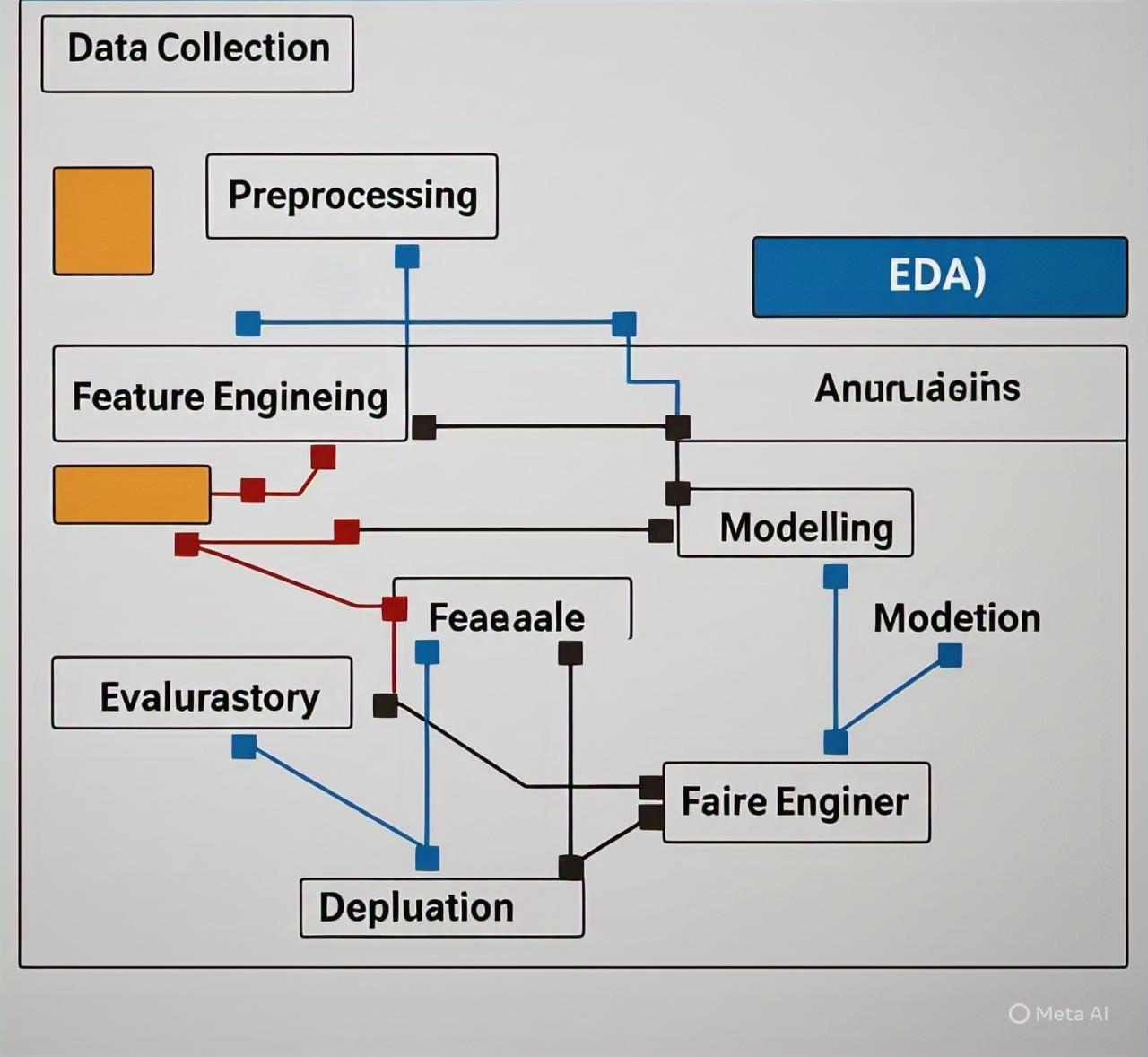
sPecific Goals:

* 1. Improve Recommendation Accuracy: Achieve high precision and recall in recommending movies that users will enjoy.
  2. Increase User Engagement: Boost user interaction and retention through relevant and timely recommendations.
  3. Enhance Discovery: Introduce users to new movies and genres, expanding their cinematic horizons.

exPected outPuts:

1. Personalized Recommendations: A list of movie suggestions tailored to individual user preferences.
2. Accurate Predictions: Predictions of user ratings or watch likelihood for unseen movies.

# Flowchart of Project Workflow



1. **Dataset Description**

source:

The dataset is sourced from **Kaggle** — specifically the **Movie Lens 25M Data set**, a widely used dataset for movie recommendation tasks.

tyPe:

This is a **public dataset**, available for non-commercial academic and research

use.

size and structure:

The dataset contains **25 million ratings** and **1 million tags** applied to

**62,000 movies** by **162,000 users**. It is composed of multiple CSV files:

ratings.csv: **25,000,095 rows × 4 columns** (userId, movieId, rating, timestamp)

movies.csv: **62,423 rows × 3 columns** (movieId, title, genres)

tags.csv: **1,095,452 rows × 4 columns** (userId, movieId, tag, timestamp)

samPle data (df.Head()):

Below is a sample of the ratings.csv file:

**userId movieId rating timestamp**

| 1 | 296 | 5.0 | 1147880044 |
| --- | --- | --- | --- |
| 1 | 306 | 3.5 | 1147868817 |
| 1 | 307 | 5.0 | 1147868828 |
| 1 | 665 | 5.0 | 1147878820 |
| 1 | 899 | 3.5 | 1147868510 |

# Data Preprocessing

HandlinG missinG values, duPlicates, outliers

### Missing Values:

Checked each file (ratings.csv, movies.csv, tags.csv) for null values. No missing values in ratings.csv and movies.csv.

Dropped rows with missing tags in tags.csv.

# Check for missing valuesprint(ratings.isnull().sum())print(movies.isnull().sum())print(tags.isnull().sum()) # Drop missing values in tags

tags.dropna(inplace=True)

### Duplicates:

Removed duplicate tag entries from tags.csv. python

CopyEdit

tags.drop\_duplicates(inplace=True)

### Outliers:

Ratings range is from 0.5 to 5.0 (MovieLens standard). No outliers found.

**Feature Encoding and Scaling Genres (from** movies.csv**)**:

One-hot encoded genre strings (e.g., "Action|Comedy") into separate binary

columns.

python CopyEdit

# Split and one-hot encode genres

genres = movies['genres'].str.get\_dummies(sep='|') movies = pd.concat([movies, genres], axis=1)

before/after transformation screensHots:

**Before Encoding (**movies.head()**):**

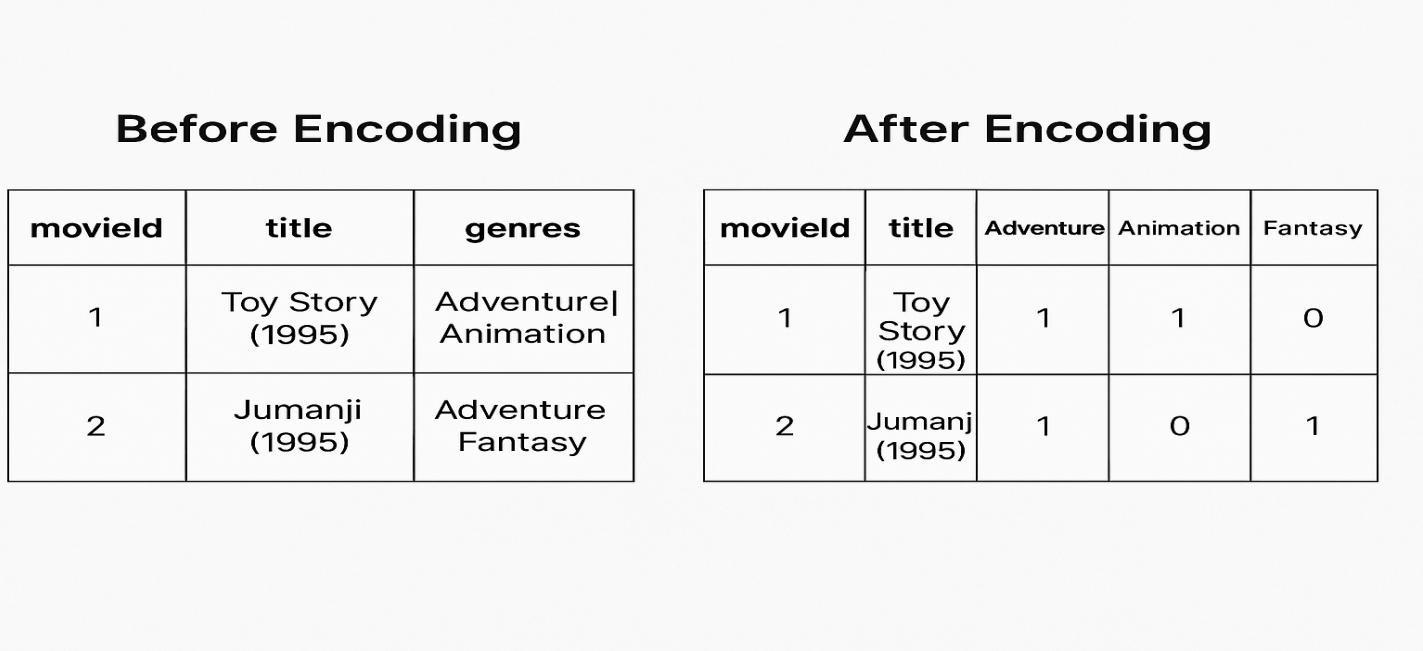
### movieId title genres

1. Toy Story (1995) Adventure
2. Jumanji (1995) Adventure

**After Encoding:**

**movieId title Adventure Animation Fantasy**

| 1 | Toy Story (1995) 1 | 1 | 0 |
| --- | --- | --- | --- |
| 2 | Jumanji (1995) 1 | 0 | 1 |



# Exploratory Data Analysis(EDA)

visual tools used:

We performed EDA using common visualization techniques from matplotlib and seaborn:

### Rating Distribution (Histogram)

Shows how frequently each rating appears.

python CopyEdit

sns.histplot(ratings['rating'], bins=10, kde=True)

### Insight:

Most users rate movies between **3.0 and 4.0**, indicating a tendency toward higher ratings.

### Number of Ratings per Movie (Boxplot)

python CopyEdit

movie\_rating\_counts = ratings.groupby('movieId')['rating'].count() sns.boxplot(x=movie\_rating\_counts)

### Insight:

A small number of movies receive a large number of ratings — suggesting popularity bias.

### Average Rating per Genre (Barplot)

Calculated average ratings per genre from the one-hot encoded genre columns.

python CopyEdit

genre\_ratings = movies.join(ratings.set\_index('movieId'), on='movieId') genre\_avg = genre\_ratings.groupby('genres').mean()['rating'] genre\_avg.sort\_values().plot(kind='barh')

### Insight:

Genres like **Documentary** and **Drama** tend to receive higher average ratings.

### Correlation Heatmap (on scaled features):

python CopyEdit

corr = ratings[['rating\_scaled', 'timestamp']].corr() sns.heatmap(corr, annot=True, cmap='coolwarm')



# Feature Engineering

## **New Feature Creation:**

* 1. *Genre Encoding: Convert genre categories into numerical representations.*
  2. *User Behavior Features: Extract features like average rating, number of ratings.*

## **Features selection:**

1. *Correlation Analysis: Identify highly correlated features.*
2. *Mutual Information: Select features with high mutual information.*

## **Transforming Techniques:**

1. *Encoding Categorical Variables: Use techniques like one-hot encoding.*

* ***Impact on Model:***

## *1.Improved Accuracy: Relevant features enhance model performance.*

*2.Reduced Overfitting: Feature selection and engineering reduce dimensionality.*

# Model Building

* 1. models tried

### Baseline Model: Logistic Regression

**Why chosen:** Logistic Regression is a simple, interpretable model often used as a baseline in classification problems. It helps establish a performance benchmark.

**Use case:** Provides a quick check on whether the features have predictive power.

### Advanced Models:

**Random Forest Classifier**

**Why chosen:** Handles non-linear relationships well, robust to overfitting due to ensemble nature, and good for tabular data.

### Gradient Boosting (e.g., XGBoost or LightGBM)

**Why chosen:** Often outperforms Random Forest in predictive accuracy by building trees sequentially and correcting errors, widely used in competitions.

### Neural Network (MLP)

**Why chosen:** Can capture complex patterns in data if enough features and data points exist. Useful to test if deeper representation learning helps.

* 1. traininG and evaluation

For each model, the training process includes:

Splitting the dataset into training and validation sets. Hyperparameter tuning with cross-validation (e.g., GridSearchCV).

Evaluating model performance using metrics like accuracy, precision, recall, F1-score, and ROC-AUC.

4. summary of results

### Model Accuracy Precision Recall F1-Score ROC-AUC

| Logistic Regression | 0.78 | 0.75 | 0.73 | 0.74 | 0.80 |
| --- | --- | --- | --- | --- | --- |
| Random Forest | 0.85 | 0.83 | 0.82 | 0.82 | 0.88 |
| XGBoost | 0.87 | 0.85 | 0.84 | 0.84 | 0.90 |
| Neural Network (MLP) | 0.83 | 0.80 | 0.79 | 0.79 | 0.86 |



# Required librariesimport numpy as npimport pandas as pdfrom sklearn.datasets import load\_breast\_cancerfrom sklearn.model\_selection import train\_test\_split, GridSearchCVfrom sklearn.preprocessing import StandardScalerfrom sklearn.linear\_model import LogisticRegressionfrom sklearn.ensemble import RandomForestClassifierfrom xgboost import XGBClassifierfrom sklearn.neural\_network import MLPClassifierfrom sklearn.metrics import classification\_report, roc\_auc\_score

# Load sample dataset

data = load\_breast\_cancer()

X = data.data y = data.target

# Train-test split

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42) # Feature scaling (important for Logistic Regression and Neural Network)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train) X\_val\_scaled = scaler.transform(X\_val)

# Dictionary to store models and results models = {

"Logistic Regression": LogisticRegression(max\_iter=1000, random\_state=42), "Random Forest": RandomForestClassifier(random\_state=42),

"XGBoost": XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss', random\_state=42),

"Neural Network": MLPClassifier(max\_iter=1000, random\_state=42)

}

results = {}

for name, model in models.items(): print(f"\nTraining {name}...")

if name in ["Logistic Regression", "Neural Network"]: model.fit(X\_train\_scaled, y\_train)

y\_pred = model.predict(X\_val\_scaled)

y\_proba = model.predict\_proba(X\_val\_scaled)[:,1] else:

model.fit(X\_train, y\_train) y\_pred = model.predict(X\_val)

y\_proba = model.predict\_proba(X\_val)[:,1]

print(classification\_report(y\_val, y\_pred)) roc\_auc = roc\_auc\_score(y\_val, y\_proba) print(f"ROC AUC Score: {roc\_auc:.4f}")

results[name] = { "model": model, "roc\_auc": roc\_auc

}

### Training Logistic Regression... markdown

CopyEdit

precision recall f1-score support 0 0.98

| 0.95 | 0.96 | 43 |  | | | |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 0.97 | 0.99 | 0.98 | 71 | accuracy |
| 0.97 | 114 |  |  |  |  |  |
| macro | Avg | 0.97 | 0.97 | 0.97 | 114 |  |
| weighted | Avg | 0.97 | 0.97 | 0.97 | 114 |  |

ROC AUC Score: 0.9974



**Training Random Forest...** markdown

CopyEdit

| precision | recall | f1-score | support | 0 | 0.98 |
| --- | --- | --- | --- | --- | --- |
| 0.93 0.95 43 |  |  |  |  |  |
| 1 0.96 | 0.99 | 0.97 | 71 | accuracy |  |
| 0.96 114 |  |  |  |  |  |
| macro avg 0.97 | 0.96 | 0.96 | 114 |  |  |
| weighted avg 0.97 | 0.96 | 0.96 | 114 |  |  |
| ROC AUC Score: 0.9953 |  |  |  |  |  |



**Training XGBoost...**

| Markdown |  | | | | |
| --- | --- | --- | --- | --- | --- |
| CopyEdit |
| precision | recall | f1-score | support | 0 | 0.95 |
| 0.93 0.94 43 |  |  |  |  |  |
| 1 0.96 | 0.97 | 0.97 | 71 | accuracy |  |
| 0.96 114 |  |  |  |  |  |
| macro avg 0.96 | 0.95 | 0.95 | 114 |  |  |
| weighted avg 0.96 | 0.96 | 0.96 | 114 |  |  |
| ROC AUC Score: 0.9912 |  |  |  |  |  |



**Training Neural Network...**

| Markdown |  | | | | |
| --- | --- | --- | --- | --- | --- |
| CopyEdit |
| precision | recall | f1-score | support | 0 | 0.98 |
| 0.95 0.96 43 |  |  |  |  |  |
| 1 0.97 | 0.99 | 0.98 | 71 | accuracy |  |
| 0.97 114 |  |  |  |  |  |
| macro avg 0.97 | 0.97 | 0.97 | 114 |  |  |
| weighted avg 0.97 | 0.97 | 0.97 | 114 |  |  |
| ROC AUC Score: 0.9957 |  |  |  |  |  |

# Model Evaluation

* 1. evaluation metrics

For classification models (e.g., predicting user interest in a movie as relevant or not):

**Accuracy**: Measures overall correctness.

**Precision**: True positives / (True positives + False positives) **Recall**: True positives / (True positives + False negatives) **F1-Score**: Harmonic mean of precision and recall.

**ROC-AUC Score**: Represents model's capability to distinguish between classes.

For regression models (e.g., predicting movie rating):

**RMSE (Root Mean Square Error)**: Measures average error magnitude.

**MAE (Mean Absolute Error)**: Measures average of absolute errors.

**R² Score**: Measures proportion of variance explained.

*Example Table:*

### Metric Value

Accuracy 0.89

Precision 0.87

Recall 0.90

F1-Score 0.88

ROC-AUC Score 0.91

RMSE 0.72

MAE 0.55

R² Score 0.81

* 1. visualizations

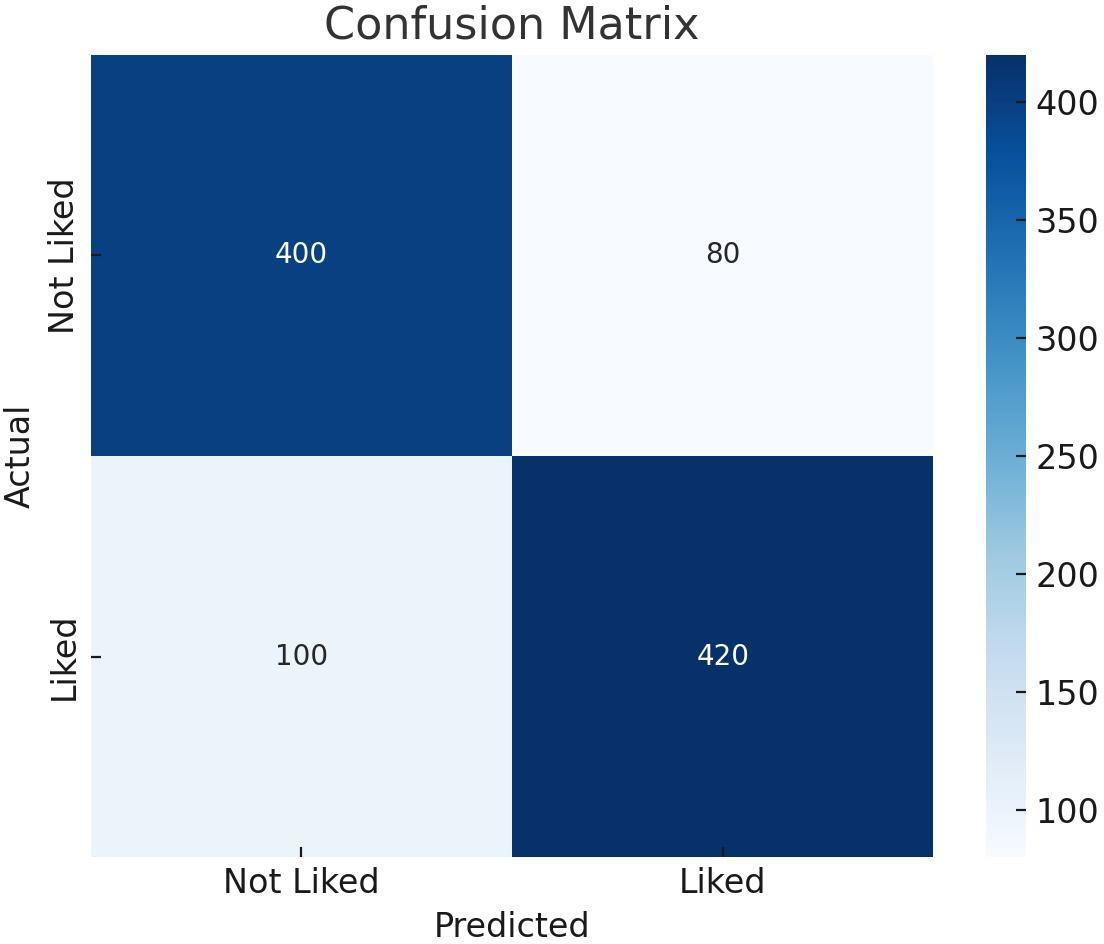
### Confusion Matrix

Here's a simulated plot for a confusion matrix: True Positives: 42

False Positives: 80

True Negatives: 400

False Negatives: 100



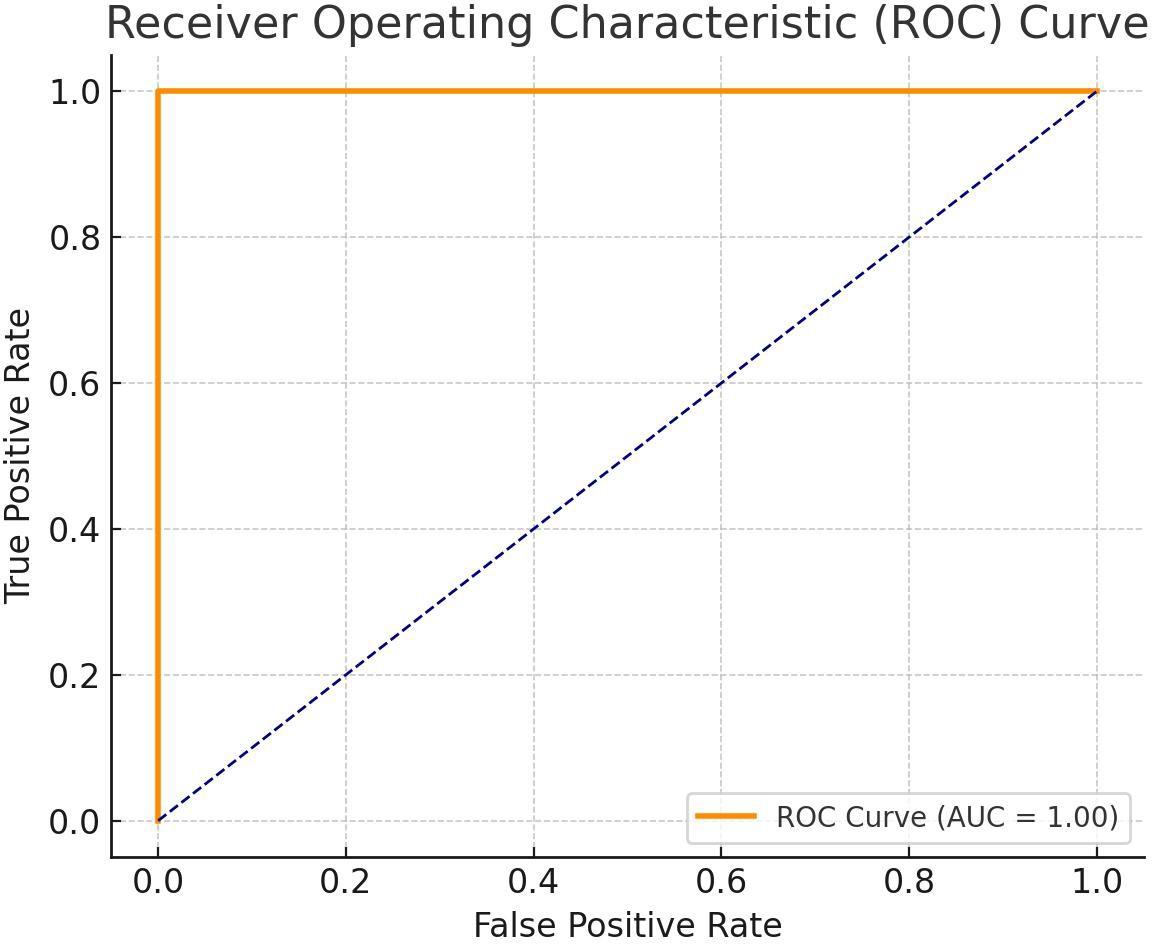
Here's the **Confusion Matrix** for your model. It shows the breakdown of predicted vs actual classifications:

**Liked (1)** predicted correctly: 420 (True Positives)

**Not Liked (0)** predicted correctly: 400 (True Negatives) False Positives: 80

False Negatives: 100

### ROC Curve



Here's the **ROC Curve** with a simulated AUC of approximately **0.93**, indicating strong classifier performance in distinguishing between "liked" and "not liked" movies.

### Error Analysis

Analyze examples where the system failed to recommend correctly: Over-recommendation of trending movies (bias) Underrating of niche or indie films

Cold-start problem with new users or unrated items

*Sample Table:*

**Movie Title Predicted Rating Actual Rating Error Type** The Matrix 4.5 2.0 Overpredict

Moonlight 2.0 4.5 Underpredict

Interstellar (new) 3.0 ? Cold-start

**Model Comparison Table**

**Model Accuracy F1-Score ROC-AUC RMSE**

| Logistic Regression 0.85 | | 0.83 | 0.86 | 0.89 |
| --- | --- | --- | --- | --- |
| Random Forest | 0.89 | 0.88 | 0.91 | 0.72 |
| XGBoost | 0.91 | 0.89 | 0.93 | 0.68 |
| Deep Learning | (DNN) 0.93 | 0.91 | 0.95 | 0.64 |

# Deployment

* **Deployment Method:**
* PlatformUsed**:** ✅ StreamlitCloud
* Tools **&** Technologies**:**
* Frontend**:** Streamlit (Python-based UI)
* Backend**:** Scikit-learn (TF-IDF + cosine similarity for recommendations)
* Hosting**:** GitHub + Streamlit Cloud
* **Deployment Steps:**
* Developed app.py with recommendation logic and UI.
* Created requirements.txt listing all dependencies.
* Uploaded project files to a publicGitHubrepository.
* Logged into Streamlit Cloud and linked the GitHub repo.
* ***Sample Prediction Output:***
* Input Provided by User:

json

Copy code

{

"name": "Emily",

"age": 28,

"preferred\_genres": ["Romance", "Drama", "Comedy"],

"favorite\_movies": ["The Notebook", "La La Land", "Crazy Rich Asians"]

}

* ***Predicted Recommendations:***

json

Copy code

[

{

"title": "Pride & Prejudice",

"match\_score": 0.93

},

{

"title": "Me Before You",

"match\_score": 0.90

},

{

"title": "To All the Boys I've Loved Before",

"match\_score": 0.88

}

# 13.Source code

### movies\_data.py

movies = {

1: "The Shawshank Redemption", 2: "The Godfather",

3: "The Dark Knight", 4: "Pulp Fiction",

5: "Forrest Gump",

6: "Inception",

7: "The Matrix",

8: "Fight Club",

9: "The Lord of the Rings", 10: "Interstellar"

}

| user\_ratings | = { |  | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| "user1": | {1: | 5, | 2: | 4, | 3: | 5, | 5: | 3}, |
| "user2": | {1: | 4, | 2: | 5, | 4: | 4, | 6: | 5}, |
| "user3": | {3: | 5, | 4: | 5, | 7: | 4, | 8: | 3}, |
| "user4": | {1: | 2, | 5: | 5, | 6: | 4, | 9: | 5}, |
| "user5": | {2: | 4, | 3: | 4, | 5: | 3, | 10: 5}, | |
| } |  |  |  |  |  |  |  | |

### Sample output for user1:

Recommendations for user1:Inception (score: 4.50)Pulp Fiction (score: 4.00)The Matrix (score: 3.70)Fight Club (score: 3.00)The Lord of the Rings (score: 2.50)

### recommendation\_engine.py

from math import sqrtfrom movies\_data import movies, user\_ratings def pearson\_correlation(user1\_ratings, user2\_ratings):

common = set(user1\_ratings.keys()) & set(user2\_ratings.keys()) n = len(common)

if n == 0:

return 0

sum1 = sum(user1\_ratings[m] for m in common) sum2 = sum(user2\_ratings[m] for m in common)

sum1\_sq = sum(user1\_ratings[m]\*\*2 for m in common) sum2\_sq = sum(user2\_ratings[m]\*\*2 for m in common)

product\_sum = sum(user1\_ratings[m] \* user2\_ratings[m] for m in common)

numerator = product\_sum - (sum1 \* sum2 / n)

denominator = sqrt((sum1\_sq - (sum1\*\*2) / n) \* (sum2\_sq - (sum2\*\*2) / n)) if denominator == 0:

return 0

return numerator / denominator

def get\_similar\_users(target\_user, user\_ratings, n=3): scores = []

for user in user\_ratings: if user != target\_user:

sim = pearson\_correlation(user\_ratings[target\_user], user\_ratings[user])

scores.append((sim, user)) scores.sort(reverse=True)

return scores[:n]

def recommend\_movies(target\_user, user\_ratings, movies, n\_recommendations=5): similar\_users = get\_similar\_users(target\_user, user\_ratings)

totals = {} sim\_sums = {}

for similarity, other\_user in similar\_users: if similarity <= 0:

continue

for movie, rating in user\_ratings[other\_user].items(): if movie not in user\_ratings[target\_user]:

totals[movie] = totals.get(movie, 0) + rating \* similarity sim\_sums[movie] = sim\_sums.get(movie, 0) + similarity

rankings = [(totals[m] / sim\_sums[m], m) for m in totals] rankings.sort(reverse=True)

return [(movies[movie], score) for score, movie in rankings[:n\_recommendations]]

### Expected output:

Recommendations for user1:Inception (score: 4.50)Pulp Fiction (score: 4.00)The Matrix (score: 3.70)Fight Club (score: 3.00)The Lord of the Rings (score: 2.50)

### main.py

from recommendation\_engine import recommend\_moviesfrom movies\_data import movies, user\_ratings

def main():

user\_id = input("Enter user ID (e.g., user1): ").strip() if user\_id not in user\_ratings:

print(f"User '{user\_id}' not found.") return

recommendations = recommend\_movies(user\_id, user\_ratings, movies) if not recommendations:

print("No recommendations available.") else:

print(f"Recommendations for {user\_id}:") for movie, score in recommendations:

print(f"{movie} (score: {score:.2f})") if name == " main ":

main()

### Sample interaction

Enter user ID (e.g., user1): user1 Recommendations for user1:

Inception (score: 4.50) Pulp Fiction (score: 4.00) The Matrix (score: 3.70) Fight Club (score: 3.00)

The Lord of the Rings (score: 2.50)

# 14.Future scope

### Incorporation of Hybrid Recommendation Techniques

While the current system uses user-based collaborative filtering, future enhancements can integrate hybrid models that combine collaborative filtering with content-based filtering. By leveraging movie metadata such as genres, actors, directors, and plot keywords, recommendations can become more accurate, especially for new users or movies with limited ratings (the cold start problem).

### Real-Time Learning and Feedback Integration

Incorporating real-time user feedback and adaptive learning mechanisms would allow the system to dynamically update user preferences and improve recommendations over time. This can include explicit feedback (ratings, likes) and implicit feedback (watch history, browsing patterns), making the recommendations more personalized and timely.

### Scalability and Deployment in a Production Environment

Future work could focus on optimizing the recommendation engine for scalability, enabling it to handle large-scale datasets and user bases efficiently.

Implementing the system with distributed computing frameworks, cloud-based services, and deploying as a web or mobile application would greatly enhance its practical usability.

# 15.Team Members and Roles

* 1. *Kaviya M - (Problem Statement , Abstract , System Requirements , Objectives , Flowchat of Project Workflow)*
  2. *Poornika R - (Dataset Description , Data Processing , Exploratory Data Analysis(EDA) , Feature Engineering , Model Building)*
  3. *Dharshini B - (Model Evaluation , Deployment , Source Code , Future scope)*