ARTIFICIAL INTELLIGENCE

(Real-World Problem Solving Using Artificial Intelligence Approaches)

Project on:

"Movie Recommendation System using IMDb dataset."

Submitted By

Yasemin Karaca (s5619032)

Submitted On

10th May 2024

Table of Content

Abstract

This report explores different movie recommendation methods, such as collaborative filtering and content-based filtering. Collaborative filtering relies on user interactions with items, whereas content-based filtering focuses on characteristics of movies like genres. Hybrid approaches combine elements of both methods to enhance recommendation accuracy. Advanced machine learning algorithms are explored to handle complex data patterns. Using a large movie dataset with ratings, the study evaluates these methods through metrics like RMSE, MAE, precision, recall, and f-1 score.

1. Introduction

As the movie industry continues to expand and it becomes more challenging for users to find movies that match their liking, relying primarily on streaming platforms is no longer sufficient to enable them to select that aligns with their preferences. A personalised movie recommendation system is suggested, utilising the extensive dataset from IMDb (Internet Movie Database), to improve user experience and engagement. Artificial intelligence (AI) has become known as one of the primary solutions to the problems with recommendation systems. AI has been effectively used in movie recommendation systems, employing advanced algorithms like deep learning to suggest films based on user preferences, enhancing user experience and system accuracy (Liu, 2023).

2. The Real-World Problem

The real-world problem is the difficulty of navigating the large and growing movie environment to discover content that meets the needs of user preferences. Without an effective recommendation system, users may spend a significant amount of time and effort sorting through numerous possibilities, resulting in dissatisfaction and possibly missing out on movies they would have enjoyed.

Addressing this problem requires the analysis of complex patterns and relationships within user preferences, viewing habits and movie characteristics. The proposed system addresses the challenge of movie discovery by applying advanced AI techniques. Specifically, machine learning algorithms are applied to analyse user behaviour, preferences, and interactions with movies, including ratings. According to Lee and Joshi (2020), AI can learn from its users' decision-making behaviours, and users should better understand how AI can support and influence their decision-making.

3. Project Aim and Objectives

The project aims to change how users find and interact with movies by utilising AI and the large amount of data in the IMDb dataset, offering a personalised and enjoyable experience.

- To design and implement a robust algorithm that utilises the IMDb dataset and can accurately suggest movies to users based on their preferences and past viewing.
- Implementing different recommendation models such as cosine similarity, K-Nearest Neighbour (KNN), Singular Value Decomposition (SVD), neural networks, hybrid approaches, content-based filtering, and decision tree algorithms to identify the most effective approach for movie suggestions.
- Designing metrics like Root Mean Square Error (RMSE), Mean Absolute Error (MAE), precision, recall and F1 score to assess the effectiveness of the models.
- Ensuring that the recommendation system is capable of handling large datasets and can effectively maintain efficient performance and response times.
- Predicting user-specific ratings for unrated movies.
- Implementing functions to generate top-n recommendations for users and personalising recommendations to their unique preferences.

4. Adopted Artificial Intelligence Approach

In the field of recommendation systems, the problem of recommending relevant and personalised movies to users has been a constant problem. Various AI approaches that integrate several machine learning models have been proposed to address this problem to deliver precise and personalised movie recommendations.

4.1. Collaborative Filtering Algorithms

Collaborative filtering generates recommendations based on the behaviour and preferences of similar users or items to make recommendations. According to the research by Melville et al. (2002), collaborative filtering has demonstrated potential effectiveness across different domains such as movie recommendations.

4.1.1. Cosine Similarity

Cosine similarity evaluates the cosine of the angle between two vectors in a multi-dimensional space to assess the similarity of users or items based on their preferences or ratings.

$$Cosine\ Similarity\ (A, B) = \frac{A.B}{\parallel A \parallel \parallel B \parallel}$$

Figure 1. Cosine Similarity Formula

According to Linden, Smith, & York (2003), this approach is particularly useful in sparse datasets common in movie recommendations, where it helps identify closely aligned preferences between users or similarity between items efficiently.

4.1.2. K-Nearest Neighbours (KNN)

KNN is a simple algorithm that recommends items by finding the nearest neighbours of a user or item based on a certain similarity metric.

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Figure 2. KNN Formula

The KNN method is clear, reliable, easy to understand, and straightforward to implement in movie recommendation systems (Adeniyi et al., 2016).

4.1.3. Singular Value Decomposition (SVD)

SVD is a matrix factorisation approach that captures implicit features related to users and items by breaking down user-item interaction matrices into lower-dimensional forms. By lowering the dataset's dimensionality, it efficiently captures user preferences and behaviours and makes it easier to find patterns in the data.

$$R \approx U\Sigma V^T$$

Where:

- U (user feature matrix) contains information about how much users "like" each feature.
- Σ (singular values matrix) is a diagonal matrix describing the strength of each latent feature.
- V^T (item-feature matrix) describes how relevant each feature is to each item.

Figure 3. SVD Formula

Paranjape et al. (2023) state that utilising SVD within collaborative filtering for movie recommendations leads to improved recommendations based on user preferences, owing to its capability to capture the fundamental traits inherent in the raw data.

4.1.4. Neural Collaborative Filtering (NCF)

Neural networks are increasingly used in recommendation systems due to their ability to handle complex, non-linear relationships between users and items. NCF is a recommendation system that relies entirely on interactions between users and items to provide recommendations (Hansel and Wibowo, 2022).

4.1.5 Recurrent Neural Network (RNN)

RNN is a neural network with nodes connected in a directional graph along a time sequence. This allows dynamic behavioural changes and the use of internal memory to process input sequences.

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Where:

- h_t = hidden state at time t
- x_t = input at time t
- W_{xh} = weights for the input layer
- W_{hh} = weights for the hidden layer
- b_h = the bias
- σ = activation function

Figure 4. RNN Formula

An RNN-based recommendation system has the advantage of being able to recognise the temporal connections in movie ratings and provide customised recommendations depending on the user's previous preferences (Labde et al., 2023)

4.2. Content-Based Filtering Algorithm

Pazzani & Billsus (2007) state that content-based filtering methods analyse the attributes or features of the items to recommend similar items to those that a user has previously liked or interacted with positively. This specific method in the notebook is customised to recommend movies based on the genre, which is a significant attribute influencing user preferences in movies.

4.3. Hybrid Recommendation System

Hybrid systems combine content-based and collaborative filtering to enhance recommendations and address issues like cold starts and scalability. They utilise movie properties and user interactions for better results.

The hybrid movie recommendation system combines collaborative and content-based filtering to successfully personalise users' movie recommendations, accomplishing low RMSE values while maintaining high precision, recall, and F1 score values (Husin et al., 2023).

4.4. Decision Tree Classifier

A Decision Tree Classifier is a machine-learning algorithm for both classification and regression challenges that can segment users or items in recommendation systems based on their characteristics.

The Decision Tree classifier is utilised in movie recommendation systems to offer customised recommendations by classifying user preferences according to movies they've previously enjoyed, demonstrating its proficiency in managing complex and varied datasets (Azaki and Baizal, 2023).

5. Artificial Intelligence Approach Implementation

Technical Problem Statement

The fundamental issue is to use past data on user interactions and movie attributes to predict user preferences and offer movies that fit their preferences.

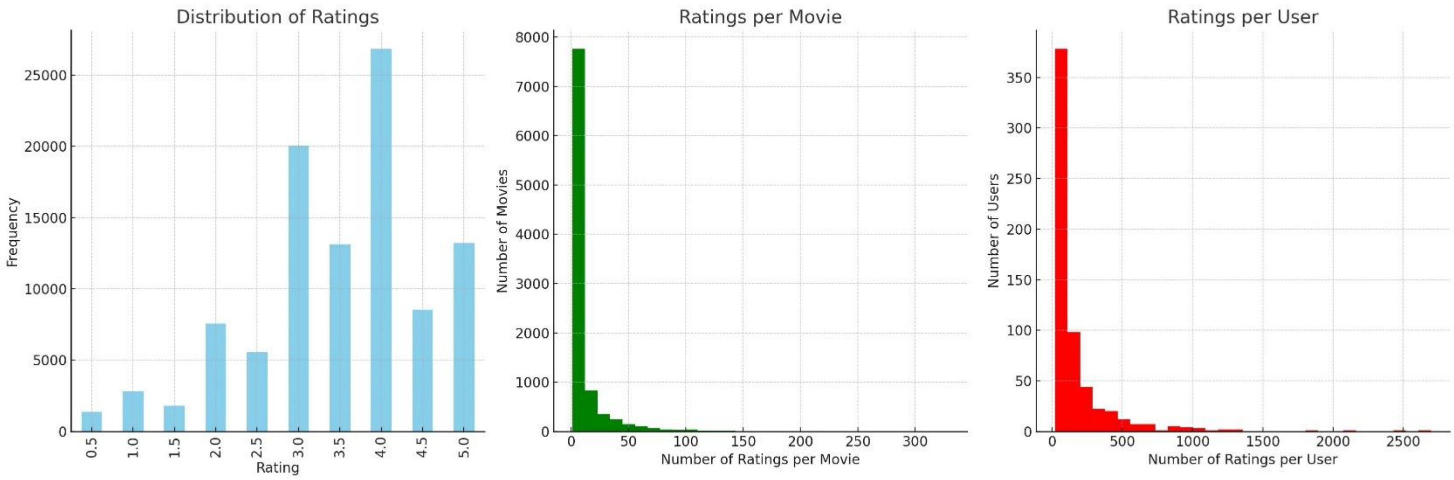


Figure 5. Distributions of User Ratings, Movie Popularity, and User Engagement

Figure 5 illustrates the challenges in creating a movie recommendation system. The first panel indicates a user bias towards higher ratings. The middle panel reveals that most movies receive few ratings, suggesting that only a select few are frequently recommended. The last panel highlights that many users rate only a few movies, complicating preference prediction due to limited data.

Hypothesis

By integrating multiple recommendation methods, we can enhance the recommendation quality by overcoming the limitations of single-model approaches like data sparsity. These AI-powered models can dynamically predict user preferences and manage complicated datasets with efficiency.

Deployment of the AI-Approach for the movie recommendation system

The process involves data preprocessing, followed by collecting user ratings and movie data to train machine learning models. These models predict user preferences by analysing user-item interactions and are refined using techniques like SVD to improve accuracy. The system then evaluates the performance and generates personalised recommendations by predicting ratings for unrated movies and suggesting the top N movies based on these predictions.

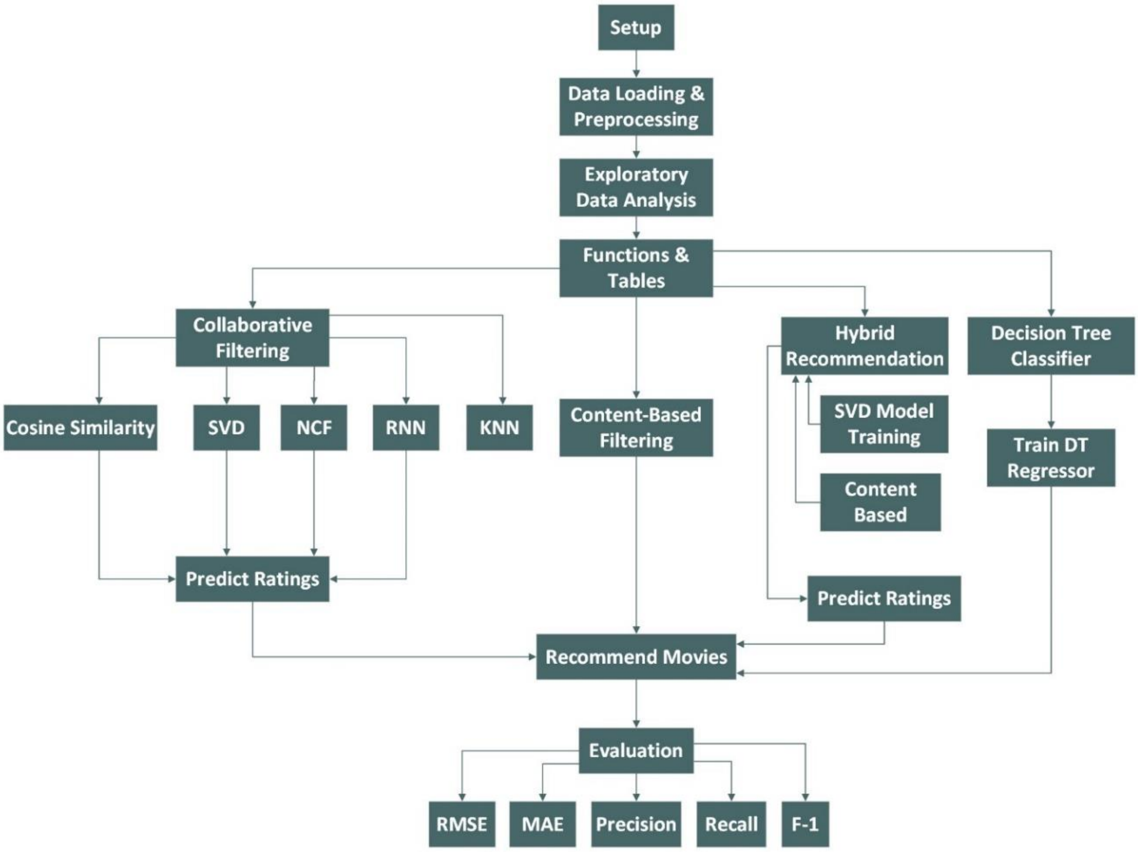


Figure 6. Structure of the code

Cosine Similarity: Calculates the cosine of the angle between two vectors in a multi-dimensional space, representing the movie ratings by different users, to determine how closely related two movies are based on user rating patterns.

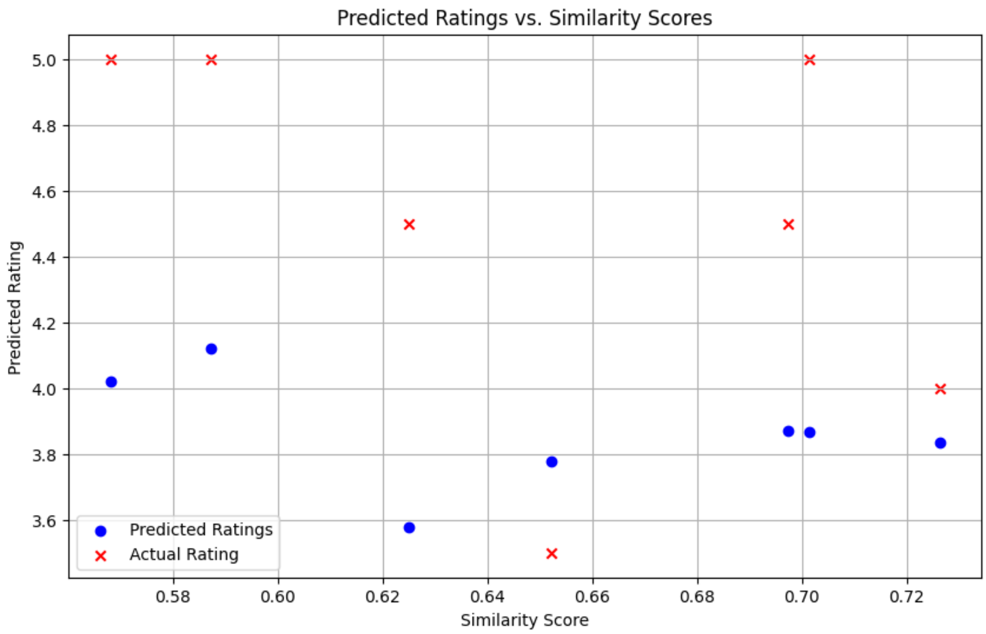


Figure 7. Cosine Similarity

SVD: Factorises the user-item rating matrix into matrices representing latent features of users and items, allowing the model to predict unknown ratings by capturing the underlying patterns in the data.

NCF: Uses a deep learning model that learns complex user-item interactions by embedding users and movies into a shared latent space and then predicting user ratings for movies based on these embeddings, offering personalised movie recommendations.

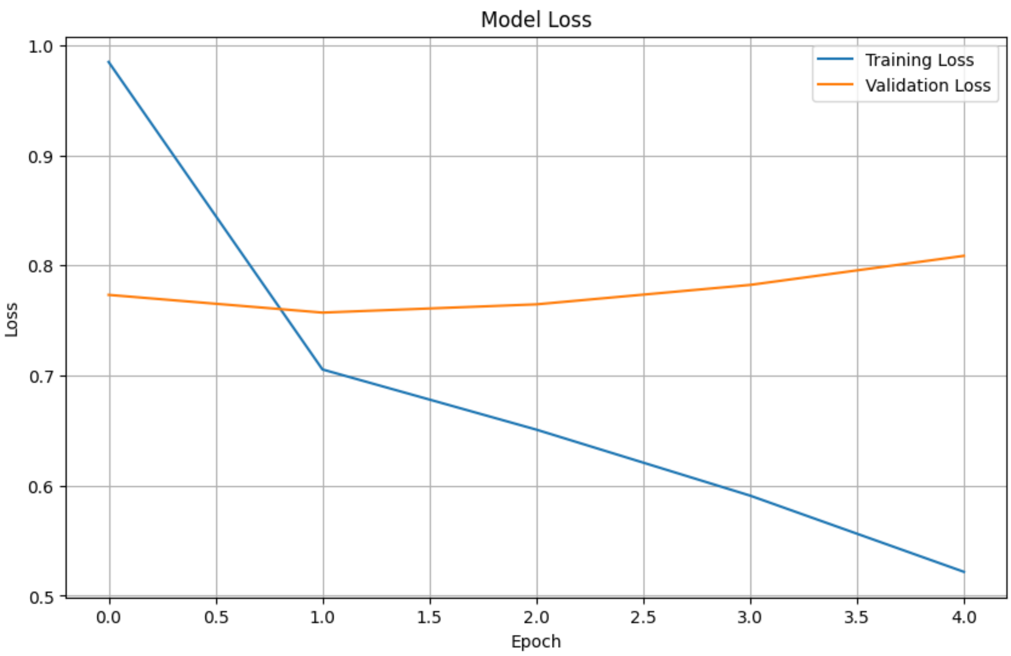


Figure 8. Training and Validation loss of NCF

The training loss decreases consistently across five epochs, indicating learning progress, while increasing validation loss after the second epoch suggests the model is overfitting and requires adjustments to improve its generalisation to new data.

RNN: Learns user preferences and movie features through embeddings, then utilises sequential processing to predict user ratings for movies, enabling personalised movie recommendations based on learned temporal patterns and interactions between users and movies.

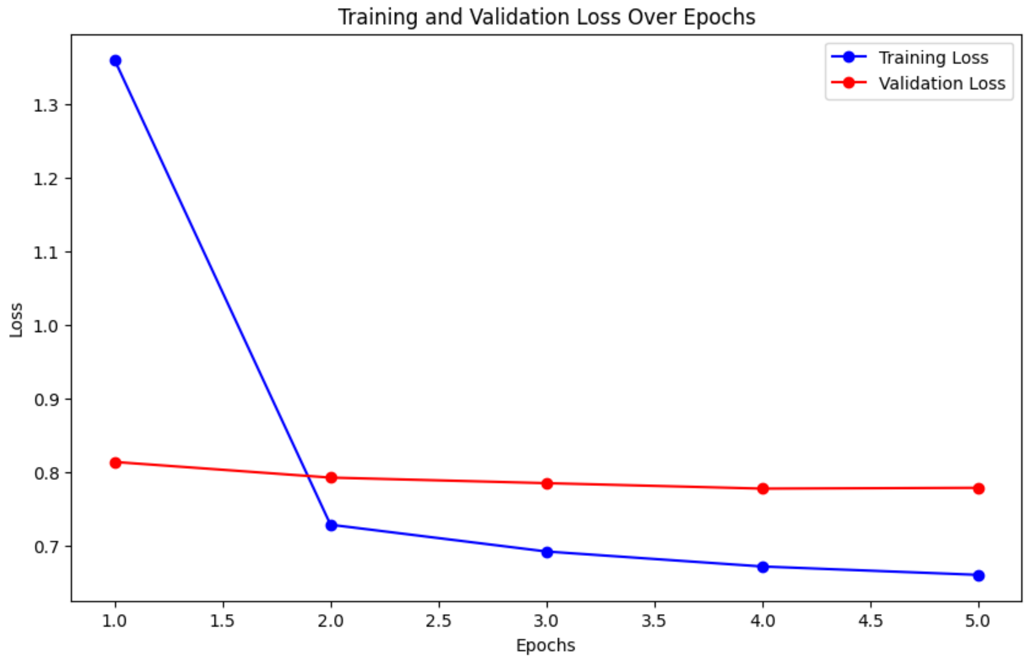


Figure 9. Training and Validation loss of RNN

KNN: Finds the top *k* movies that are most similar to a given movie based on cosine similarity in the user-item rating matrix, it provides recommendations by identifying movies with similar user rating patterns to the target movie.

Content-Based Filtering: Utilises a TF-IDF vectorisation of movie genres to create feature vectors, computes the cosine similarity between these vectors to identify similar movies, and recommends movies by matching genres, thus personalising suggestions based on specific content attributes of the movies.

Hybrid Recommendation: Combines the SVD model with content-based filtering to suggest movies within a user's preferred genre that they have not yet rated, using collaborative filtering to predict how much they might like them.

Recommended Movies for the chosen genre:

	Title	Predicted Rating
0	Star Wars: Episode IV - A New Hope (1977)	4.837075
1	Eternal Sunshine of the Spotless Mind (2004)	4.727848
2	Star Wars: Episode V - The Empire Strikes Back...	4.703778
3	Terminator 2: Judgment Day (1991)	4.698447
4	The Martian (2015)	4.691773
5	Logan (2017)	4.662412
6	Brazil (1985)	4.639488
7	Grand Day Out with Wallace and Gromit, A (1989)	4.627940
8	WALL·E (2008)	4.626973
9	Serenity (2005)	4.612020

Figure 10. Example Output of Hybrid Recommendation

Decision Tree Classifier: Learns from user-item interactions and other features to predict whether a user would like a movie based on their past ratings, and it uses these predictions to recommend unseen movies to users.

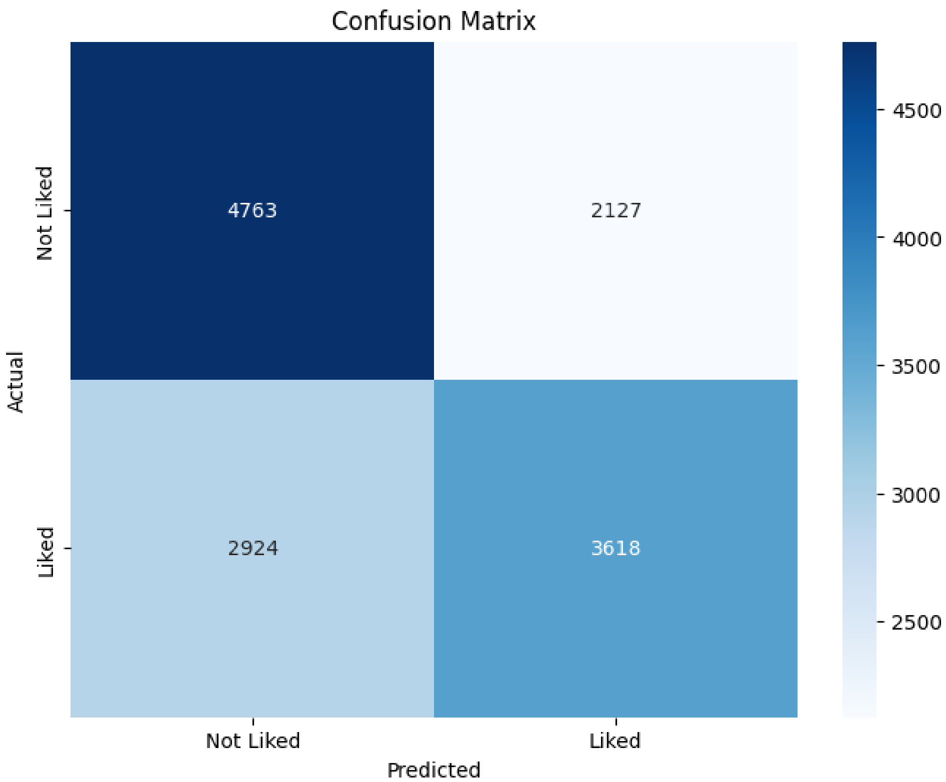


Figure 11. Confusion Matrix

It is seen that the model correctly predicted 3,618 likes and 4,763 dislikes but misclassified 2,924 likes as dislikes and 2,127 dislikes as likes, showing challenges in accurately distinguishing user preferences.

6. Evaluation, Results and Discussions

The effectiveness of movie recommendation models was assessed using key metrics. These measurements provide information on the accuracy and feasibility of each model.

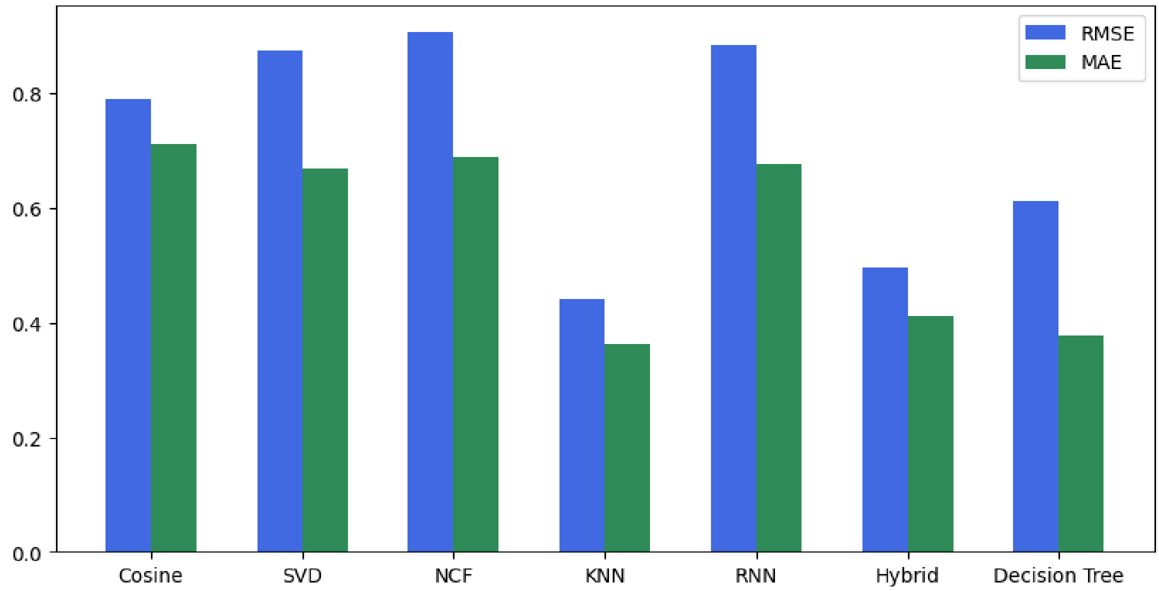


Figure 12. Comparison of RMSE and MAE across Models

Various example usages were provided to the methods. From the results, the KNN and Hybrid models shows lower RMSE and MAE, indicating better accuracy in predicting ratings. RNN and NCF shows higher errors, this might be due to overfitting.

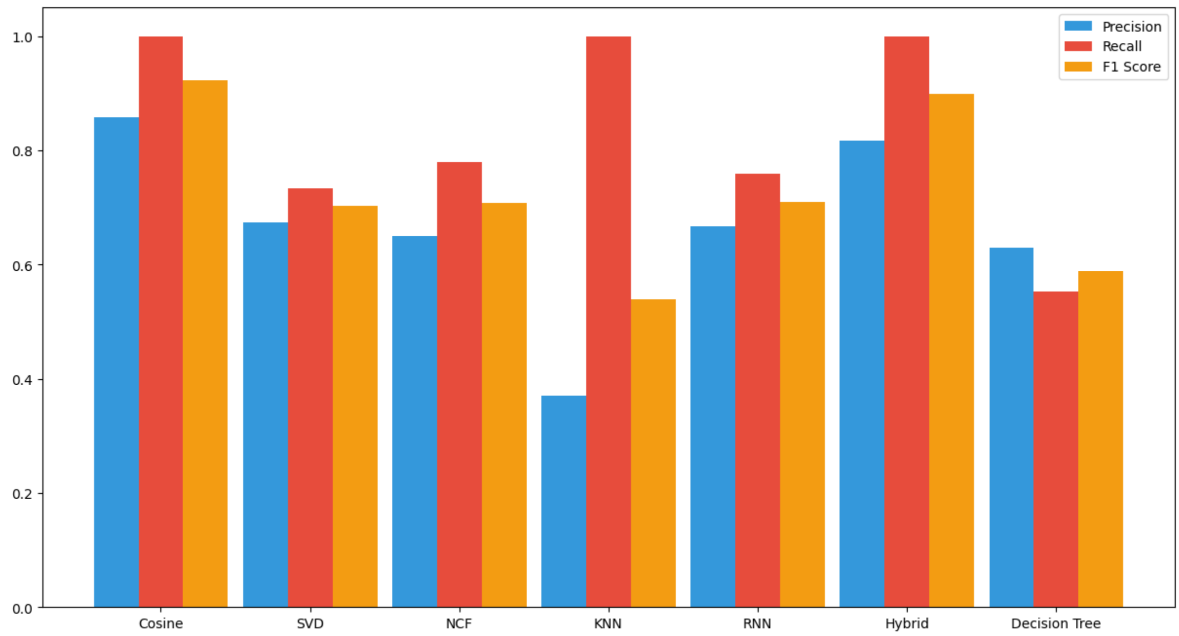


Figure 13. Comparison of Model Metrics

Precision, recall and F1 scores are useful to evaluate models based on the relevancy of the recommended movie. It is seen that Hybrid perform better and balanced, meaning it is effective at identifying relevant items without many false positives. The Decision Tree model lagged in these metrics, indicating a potential mismatch between the model complexity and the data complexity.

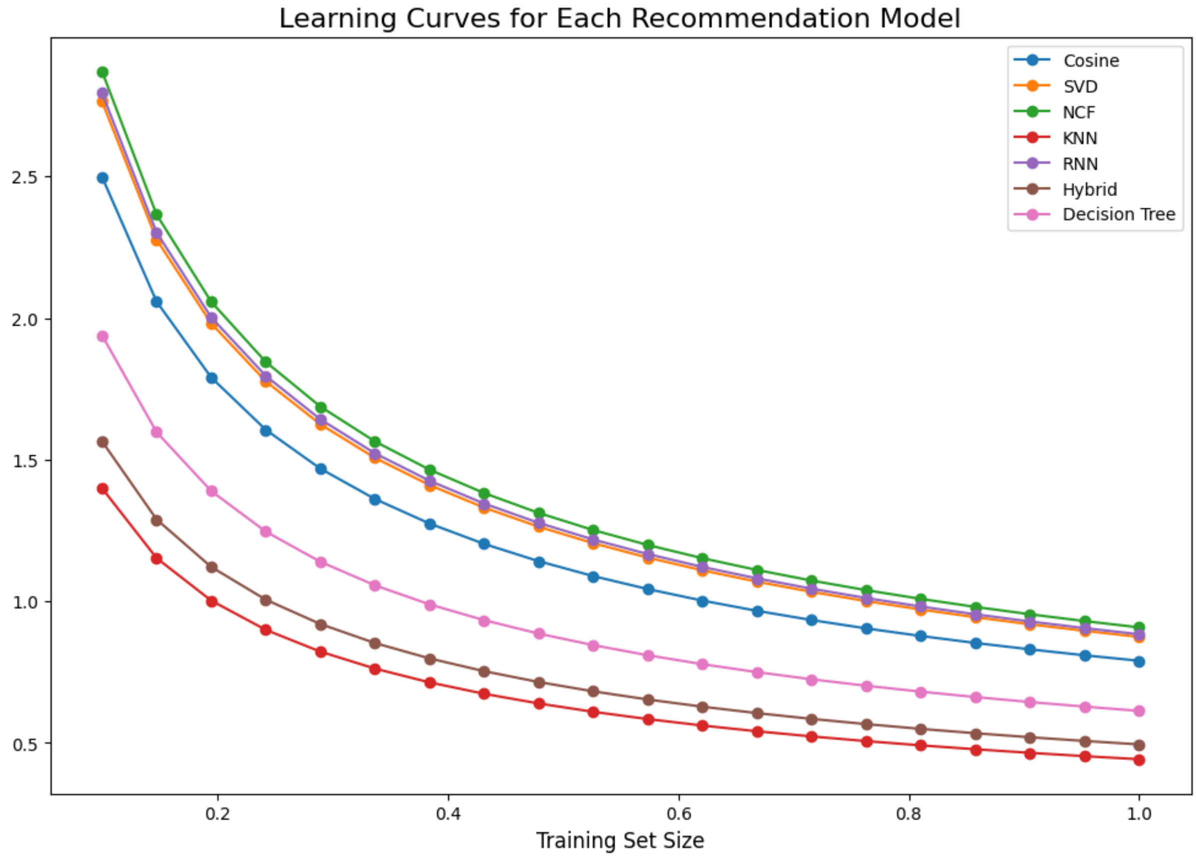


Figure 14. Learning Curves for Each Recommendation Model

The learning curves illustrate how different recommendation models adapt with increased training data. Hybrid and KNN models show gradual improvements, indicating moderate learning rates. In contrast, SVD, NCF, and RNN models display a rapid reduction in error rates, but they require significant data to achieve lower errors, suggesting higher computational costs and longer training times. This might limit scalability in real-time systems with large, growing databases. Simpler models like Decision Trees provide quicker execution but may lack accuracy with complex data.

Choosing a model for deployment involves balancing accuracy with computational demands. For real-time recommendations, scalable models like KNN, potentially enhanced by efficient indexing, are preferable. For requiring high accuracy and the capability to manage diverse and complex user preferences, the hybrid model is optimal. Implementing matrix factorisation techniques could improve the scalability and efficiency of computationally intensive models like SVD, NCF and RNN.

7. Conclusion and Future Work

This project successfully developed AI-based algorithms to improve movie recommendations on streaming platforms, using various approaches. It enhanced accuracy and user satisfaction by addressing data sparsity and scalability issues with advanced models. The hybrid approach achieved lower error rates and higher accuracy in evaluations compared to other models, aiming to continually improve the movie discovery experience. Additionally, exploring user feedback mechanisms to refine the system's learning algorithms will ensure the system remains adaptive and efficient in meeting changing user preferences.

8. References

1. Adeniyi, D. A., Wei, Z. and Yongquan, Y., 2016. Automated web usage data mining and recommendation system using K-Nearest Neighbor (KNN) classification method. Applied Computing and Informatics, 12 (1), 90–108.

2. Adomavicius, G. and Tuzhilin, A., 2005. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. IEEE Transactions on Knowledge and Data Engineering, 17 (6), 734–749.

3. Azaki, M. B. R. and Baizal, Z. K. A., 2023. Movie Recommender System Using Decision Tree Method. JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika) [online], 8 (3). Available from: https://scholar.archive.org/work/hvykvxkrujgc7fkbxfdoxxuk7y [Accessed 26 Apr 2024].

4. Hansel, A. C. and Wibowo, A., 2022. Using Movie Genres in Neural Network Based Collaborative Filtering Movie Recommendation System to Reduce Cold Start Problem. International Journal of Emerging Technology and Advanced Engineering, 12 (3), 63–73.

5. Jeong, I.-Y., Yang, X. and Jung, H.-K., 2015. A Study on Movies Recommendation System of Hybrid Filtering-Based. Journal of the Korea Institute of Information and Communication Engineering, 19 (1), 113–118.

6. Kumar, N. P. and Fan, Z., 2015. Hybrid User-Item Based Collaborative Filtering. Procedia Computer Science, 60, 1453–1461.

7. Labde, S., Karan, V., Shah, S. and Krishnan, D., 2023. Movie Recommendation System using RNN and Cognitive thinking [online]. IEEE. Available from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10170572.

8. Lee, K. and Joshi, K., 2020. Understanding the Role of Cultural Context and User Interaction in Artificial Intelligence Based Systems. Journal of Global Information Technology Management, 23 (3), 171–175.

9. Liu, L., 2023. Deep learning methods used in movie recommendation systems. Applied and Computational Engineering, 15 (1), 149–154.

10. Lops, P., Jannach, D., Musto, C., Bogers, T. and Koolen, M., 2019. Trends in content-based recommendation. User Modeling and User-Adapted Interaction, 29 (2), 239–249.

11. Melville, P., Mooney, R. and Nagarajan, R., 2002. Content-Boosted Collaborative Filtering for Improved Recommendations [online]. Available from: https://www.cs.utexas.edu/~ml/papers/cbcf-aaai-02.pdf.

12. Omega, C. Z. and Hendry, H., 2021. Movie Recommendation System using Weighted Average Approach [online]. Available from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9590147.

13. Paranjape, V., Nihalani, N. and Mishra, N., 2023. Design of a Hybrid Movie Recommender System Using Machine Learning. International Journal of Emerging Technology and Advanced Engineering [online], 13 (3), 159–165. Available from: https://ijetae.com/files/Volume13Issue3/IJETAE_0323_17.pdf [Accessed 26 Apr 2024].

14. Pazzani, M. J. and Billsus, D., 2007. Content-Based Recommendation Systems. The Adaptive Web, 4321, 325–341.

15. Rai, A., Yadav, K., Singh, M. and Singh, S. K., 2022. Accuracy Comparison of Various Movie Recommendation System Algorithms. 2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N) [online], 254–260. Available from: https://www.semanticscholar.org/paper/Accuracy-Comparison-of-Various-Movie-Recommendation-Rai-Yadav/a46ea62003b735036ec505cd3f3d3d67e76e5f52 [Accessed 28 Apr 2024].

16. Schein, A. I., Popescul, A., Ungar, L. H. and Pennock, D. M., 2002. Methods and metrics for cold-start recommendations. Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR ’02, 25 (1).

9. Appendix

Appendix A. Setup and Functions

Setup and Functions

+ Code + Text

```python
def load_data():
    ratings = pd.read_csv('ratings.csv')
    movies = pd.read_csv('movies.csv')
    return ratings, movies
```

```python
def create_matrices(ratings, movies):
    merged_df = pd.merge(ratings, movies, on='movieId')
    user_item_matrix = merged_df.pivot_table(index='userId', columns='title', values='rating', fill_value=0)
    cosine_sim = cosine_similarity(user_item_matrix.T)
    return user_item_matrix, cosine_sim
```

```python
def get_user_ratings(user_item_matrix, user_id):
    user_ratings = user_item_matrix.loc[user_id]
    rated_movies = user_ratings[user_ratings > 0].index.tolist()
    return user_ratings, rated_movies
```

```python
def get_movie_index(user_item_matrix, movie_title):
    return user_item_matrix.columns.get_loc(movie_title) if movie_title in user_item_matrix.columns else -1
```

```python
def calculate_metrics(actual_ratings, predicted_ratings):

    filtered_ratings = [(actual, predicted) for actual, predicted in zip(actual_ratings, predicted_ratings) if predicted is not None]
    if not filtered_ratings:
        return float('nan'), float('nan')

    actuals, estimates = zip(*filtered_ratings)
    rmse = sqrt(mean_squared_error(actuals, estimates))
    mae = mean_absolute_error(actuals, estimates)
    return rmse, mae
```

```python
def calculate_classification_metrics(actuals, predictions, threshold=3.5):
    actual_classes = [1 if x > threshold else 0 for x in actuals]
    predicted_classes = [1 if x > threshold else 0 for x in predictions]

    tp = sum((ac == 1) and (pc == 1) for ac, pc in zip(actual_classes, predicted_classes))
    tn = sum((ac == 0) and (pc == 0) for ac, pc in zip(actual_classes, predicted_classes))
    fp = sum((ac == 0) and (pc == 1) for ac, pc in zip(actual_classes, predicted_classes))
    fn = sum((ac == 1) and (pc == 0) for ac, pc in zip(actual_classes, predicted_classes))

    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    return {"Precision": precision, "Recall": recall, "F1-Score": f1}
```

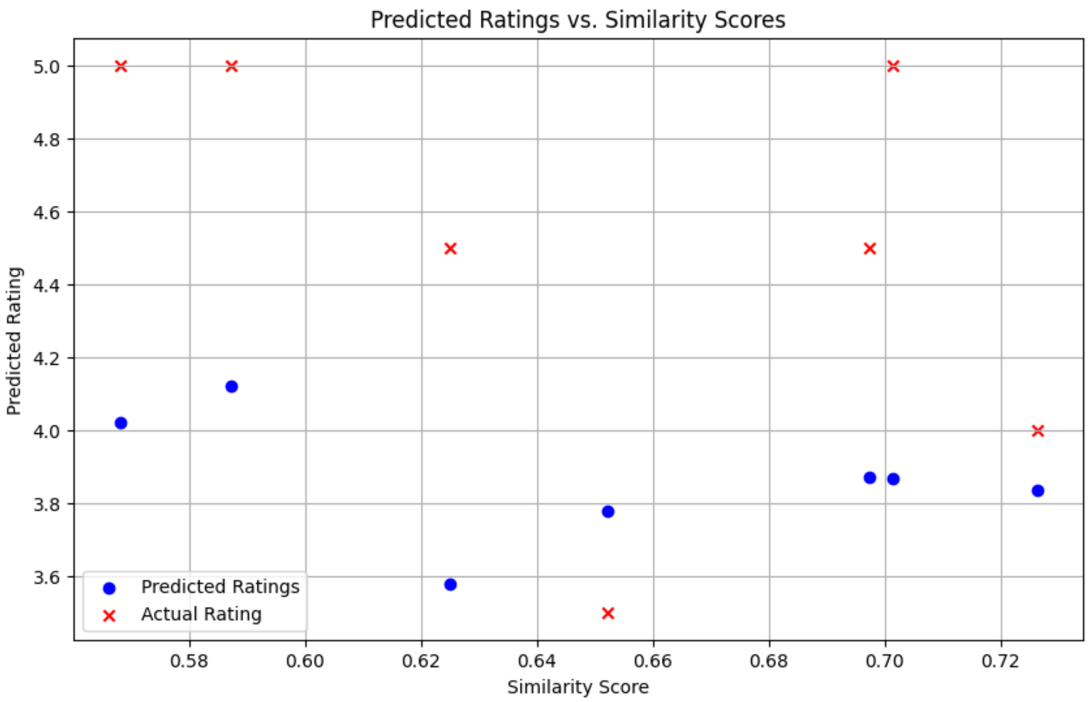
Appendix B. Cosine Similarity

Cosine Similarity

```
def recommend_movies_cosine(user_item_matrix, cosine_sim, user_id, movie_title, k=10):
    movie_idx = get_movie_index(user_item_matrix, movie_title) # get the index of the movie in the user item matrix
    similar_movies = sorted(list(enumerate(cosine_sim[movie_idx])), key=lambda x: x[1], reverse=True)
    # sort movies based on their cosine similarity with the target movie
    recommendations = [(user_item_matrix.columns[i], score) for i, score in similar_movies[1:k+1]]

    user_ratings = user_item_matrix.loc[user_id]
    predictions = [] # predicting ratings for recommended movies
    for movie, score in recommendations:
        similar_user_ratings = user_item_matrix[movie] # get ratings of users who rated the recommended movie
        if (similar_user_ratings != 0).sum() > 0:
            weighted_ratings = similar_user_ratings * score  # calculate predicted rating using weighted average of similar user's ratings
            predicted_rating = weighted_ratings.sum() / (score * (similar_user_ratings != 0).sum())
        else:
            predicted_rating = user_ratings.mean()  # if no similar users have rated the movie, predict the average rating of the user
        predictions.append((movie, predicted_rating, score))
    return predictions
```

```
ratings, movies = load_data()
user_item_matrix, cosine_sim = create_matrices(ratings, movies)

# example usage
movie_title = 'Finding Nemo (2003)'
user_id = 66

predictions = recommend_movies_cosine(user_item_matrix, cosine_sim, user_id, movie_title) # fetching predictions

if predictions:  # analysing predictions
    print("Predicted ratings and similarity scores for recommended movies:")
    for movie, predicted_rating, score in predictions:
        print(f"{movie}: Predicted Rating = {predicted_rating:.4f}, Similarity Score = {score:.4f}")

    # comparison of actual and predicted ratings
    actual_ratings = []
    predicted_ratings = []
    similarity_scores = []
    for movie, predicted_rating, similarity_score in predictions:
        if user_item_matrix.at[user_id, movie] > 0:  # Checking if the user has rated the movie
            actual_ratings.append(user_item_matrix.at[user_id, movie])
            predicted_ratings.append(predicted_rating)
            similarity_scores.append(similarity_score)

    # calculation of metrics
    if actual_ratings:
        rmse, mae = calculate_metrics(actual_ratings, predicted_ratings)
        print(f"RMSE: {rmse:.4f}, MAE: {mae:.4f}")

        metrics = calculate_classification_metrics(actual_ratings, predicted_ratings)
        print(f"Precision: {metrics['Precision']:.4f}")
        print(f"Recall: {metrics['Recall']:.4f}")
        print(f"F1-Score: {metrics['F1-Score']:.4f}")

        # create plot
        plt.figure(figsize=(10, 6))
        plt.scatter(similarity_scores, predicted_ratings, color='blue', label='Predicted Ratings')
        for i, actual_rating in enumerate(actual_ratings):
            plt.scatter(similarity_scores[i], actual_rating, color='red', marker='x', label='Actual Rating' if i == 0 else "")

        plt.title('Predicted Ratings vs. Similarity Scores')
        plt.xlabel('Similarity Score')
        plt.ylabel('Predicted Rating')
        plt.legend()
        plt.grid(True)
        plt.show()
```

```
Predicted ratings and similarity scores for recommended movies:
Incredibles, The (2004): Predicted Rating = 3.8360, Similarity Score = 0.7264
Shrek (2001): Predicted Rating = 3.8676, Similarity Score = 0.7014
Monsters, Inc. (2001): Predicted Rating = 3.8712, Similarity Score = 0.6973
Pirates of the Caribbean: The Curse of the Black Pearl (2003): Predicted Rating = 3.7785, Similarity Score = 0.6523
Shrek 2 (2004): Predicted Rating = 3.5761, Similarity Score = 0.6251
Catch Me If You Can (2002): Predicted Rating = 3.9217, Similarity Score = 0.5942
Lord of the Rings: The Return of the King, The (2003): Predicted Rating = 4.1189, Similarity Score = 0.5873
Lord of the Rings: The Two Towers, The (2002): Predicted Rating = 4.0213, Similarity Score = 0.5681
Harry Potter and the Prisoner of Azkaban (2004): Predicted Rating = 3.9140, Similarity Score = 0.5603
Ocean's Eleven (2001): Predicted Rating = 3.8445, Similarity Score = 0.5568
RMSE: 0.7901, MAE: 0.7125
Precision: 0.8571
Recall: 1.0000
F1-Score: 0.9231
```



Appendix C. SVD

Singular Value Decomposition (SVD)

```
def svd_process(ratings):
    reader = Reader(rating_scale=(0.5, 5))
    data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
    trainset, testset = surprise_train_test_split(data, test_size=0.2)  # split the data into training and testing sets
    model = SVD()
    model.fit(trainset) # train the SVD model on the training set
    predictions = model.test(testset)
    return model, predictions
```

```
[147] def get_top_n_recommendations(model, ratings, movies, user_id, n=10):
    user_movies = ratings[ratings['userId'] == user_id]['movieId'] # get movies that user has already rated
    user_unrated_movies = movies[~movies['movieId'].isin(user_movies)]['movieId']
    predictions = [(movie_id, model.predict(user_id, movie_id).est) for movie_id in user_unrated_movies]
    top_n = sorted(predictions, key=lambda x: x[1], reverse=True)[:n] # sort predictions by estimated rating in descending order and select top n
    top_n_movies = [(movie_info[0], movies[movies['movieId'] == movie_info[0]]['title'].iloc[0], movie_info[1]) for movie_info in top_n]
    return top_n_movies
```

```
[148] svd_model, svd_predictions = svd_process(ratings)

user_id = 15
top_svd_recommendations = get_top_n_recommendations(svd_model, ratings, movies, user_id)
print("\nTop SVD Recommendations:")
for movie_id, title, predicted_rating in top_svd_recommendations:
    print(f"Movie ID: {movie_id} | Title: {title} | Predicted Rating: {predicted_rating:.2f}")
```

```
Top SVD Recommendations:
Movie ID: 1223 | Title: Grand Day Out with Wallace and Gromit, A (1989) | Predicted Rating: 4.33
Movie ID: 1193 | Title: One Flew Over the Cuckoo's Nest (1975) | Predicted Rating: 4.27
Movie ID: 48516 | Title: Departed, The (2006) | Predicted Rating: 4.20
Movie ID: 1213 | Title: Goodfellas (1990) | Predicted Rating: 4.20
Movie ID: 110 | Title: Braveheart (1995) | Predicted Rating: 4.19
Movie ID: 720 | Title: Wallace & Gromit: The Best of Aardman Animation (1996) | Predicted Rating: 4.17
Movie ID: 1104 | Title: Streetcar Named Desire, A (1951) | Predicted Rating: 4.16
Movie ID: 1953 | Title: French Connection, The (1971) | Predicted Rating: 4.15
Movie ID: 1201 | Title: Good, the Bad and the Ugly, The (Buono, il brutto, il cattivo, Il) (1966) | Predicted Rating: 4.15
Movie ID: 3681 | Title: For a Few Dollars More (Per qualche dollaro in più) (1965) | Predicted Rating: 4.14
```

```
[149] actual_ratings = [pred.r_ui for pred in svd_predictions] # extracting actual ratings from the SVD predictions
predicted_ratings = [pred.est for pred in svd_predictions if pred.est is not None] # extracting predicted ratings
rmse, mae = calculate_metrics(actual_ratings, predicted_ratings)
print(f"RMSE: {rmse:.4f}, MAE: {mae:.4f}")
```

```
RMSE: 0.8743, MAE: 0.6698
```

```
[150] svd_classification_predictions = [(pred.r_ui, pred.est) for pred in svd_predictions if pred.est is not None]

actuals, predictions = zip(*svd_classification_predictions)  # unpack the list of tuples into separate lists for actuals and predictions
svd_metrics = calculate_classification_metrics(actuals, predictions)
print(f"Precision: {svd_metrics['Precision']:.4f}\nRecall: {svd_metrics['Recall']:.4f}\nF1-Score: {svd_metrics['F1-Score']:.4f}")
```

```
Precision: 0.6740
Recall: 0.7330
F1-Score: 0.7022
```

Appendix D. NCF

Neural Collaborative Filtering(NCF)

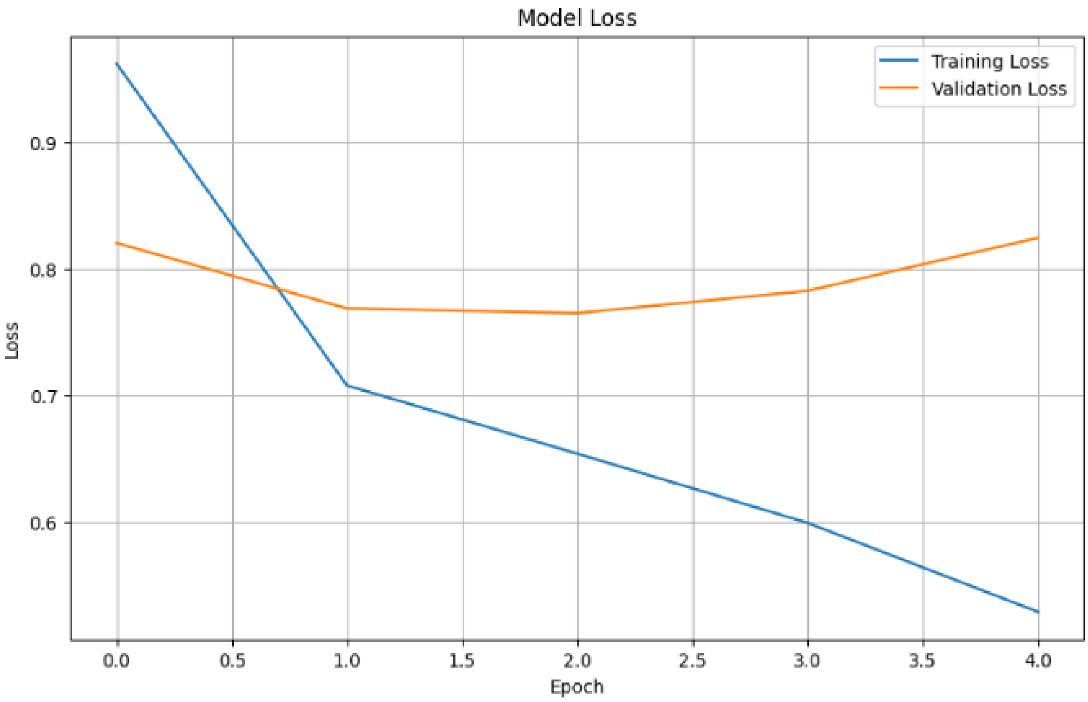
```
def build_and_train_ncf_model(ratings):
    user_encoder = LabelEncoder() # initialising label encoders for user and movie
    movie_encoder = LabelEncoder()

    ratings['userId'] = user_encoder.fit_transform(ratings['userId'])
    ratings['movieId'] = movie_encoder.fit_transform(ratings['movieId'])

    train, test = train_test_split(ratings, test_size=0.2, random_state=42) # split data into train and test sets
    num_users = ratings['userId'].nunique() # get unique number of users and movies
    num_movies = ratings['movieId'].nunique()

    embedding_size = 50 # setting embedding size

    user_input = Input(shape=(1,), name='user_input') # defining input layers for user and movie IDs
    movie_input = Input(shape=(1,), name='movie_input')

    user_embedding = Embedding(num_users, embedding_size, name='user_embedding')(user_input)
    movie_embedding = Embedding(num_movies, embedding_size, name='movie_embedding')(movie_input)

    user_vec = Flatten(name='flatten_users')(user_embedding)
    movie_vec = Flatten(name='flatten_movies')(movie_embedding)

    concat = Concatenate()([user_vec, movie_vec])# concatenate user and movie vectors

    dense = Dense(128, activation='relu')(concat)
    dense = Dense(64, activation='relu')(dense)
    outputs = Dense(1, activation='linear')(dense)

    model = Model(inputs=[user_input, movie_input], outputs=outputs)    # define and compile the model
    model.compile(optimizer=Adam(0.001), loss='mean_squared_error')

    history = model.fit( # train the model
        [train['userId'].values, train['movieId'].values],
        train['rating'].values,
        batch_size=32,
        epochs=5,
        validation_data=(
            [test['userId'].values, test['movieId'].values],
            test['rating'].values
        )
    )

    plt.figure(figsize=(10, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```python
def recommend_movies_ncf(model, ratings, movies, user_id, user_encoder, movie_encoder, top_n=10):
    user_idx = user_encoder.transform([user_id])[0] # converting user id to its encoded form
    rated_user_movies = ratings[ratings['userId'] == user_id]['movieId'].unique() # finding movies already rated by the user

    all_movie_ids = movies['movieId'].unique()
    movies_to_predict = np.setdiff1d(all_movie_ids, rated_user_movies)

    valid_movie_ids = [mid for mid in movies_to_predict if mid in movie_encoder.classes_]  # filtering out movie ids that are not in the movie encoder's classes

    user_idx_array = np.array([user_idx] * len(valid_movie_ids))  # repeat user index for each valid movie id
    movie_idx_array = movie_encoder.transform(valid_movie_ids)  # encode valid movie ids

    predictions = model.predict([user_idx_array, movie_idx_array]).flatten()
    predictions = np.clip(predictions, 0.5, 5)

    top_n_indices = predictions.argsort()[-top_n:][::-1]
    recommended_movie_ids = np.array(valid_movie_ids)[top_n_indices]
    recommended_movie_scores = predictions[top_n_indices]

    recommended_movies = movies[movies['movieId'].isin(recommended_movie_ids)]
    recommended_movies = recommended_movies.copy()
    recommended_movies.loc[:, 'predicted_rating'] = recommended_movie_scores

    recommended_movies = recommended_movies.sort_values('predicted_rating', ascending=False)

    return recommended_movies[['title', 'predicted_rating']]
```

[153]
```python
def prepare_ncf_predictions(model, test, user_encoder, movie_encoder):
    test_predictions = model.predict([test['userId'].values, test['movieId'].values]).flatten()
    actual_predicted = list(zip(test['rating'].values, test_predictions))  # combining actual and predicted ratings
    return actual_predicted
```

[154]
```python
ncf_model, ncf_rmse, ncf_mae, user_encoder, movie_encoder, test = build_and_train_ncf_model(ratings)
print(f"\nRMSE: {ncf_rmse:.4f}\nMAE: {ncf_mae:.4f}")
print("NCF Model Trained")
```

```
Epoch 1/5
2521/2521 [==============================] - 39s 14ms/step - loss: 0.9621 - val_loss: 0.8206
Epoch 2/5
2521/2521 [==============================] - 26s 10ms/step - loss: 0.7081 - val_loss: 0.7689
Epoch 3/5
2521/2521 [==============================] - 25s 10ms/step - loss: 0.6543 - val_loss: 0.7651
Epoch 4/5
2521/2521 [==============================] - 26s 10ms/step - loss: 0.5996 - val_loss: 0.7828
Epoch 5/5
2521/2521 [==============================] - 27s 11ms/step - loss: 0.5294 - val_loss: 0.8245
```




```
631/631 [==============================] - 1s 1ms/step

RMSE: 0.9080
MAE: 0.6882
NCF Model Trained
```

```
[248] user_id = 10
      ncf_recommendations = recommend_movies_ncf(ncf_model, ratings, movies, user_id, user_encoder, movie_encoder)
      print("NCF Recommendations:")
      display(ncf_recommendations)

      ncf_predictions = prepare_ncf_predictions(ncf_model, test, user_encoder, movie_encoder)
      actuals, predictions = zip(*ncf_predictions)

      ncf_classification_metrics = calculate_classification_metrics(actuals, predictions, threshold = 3.5)
      print(f"Precision: {ncf_classification_metrics['Precision']:.4f}")
      print(f"Recall: {ncf_classification_metrics['Recall']:.4f}")
      print(f"F1-Score: {ncf_classification_metrics['F1-Score']:.4f}")
```

```
303/303 [==============================] - 1s 3ms/step
NCF Recommendations:
```

	title	predicted_rating
62	From Dusk Till Dawn (1996)	5.000000
592	Rock, The (1996)	5.000000
1714	Nashville (1975)	4.976501
1961	eXistenZ (1999)	4.829728
2959	Billy Elliot (2000)	4.826491
3734	Hangar 18 (1980)	4.773508
5580	Bad Boy Bubby (1993)	4.769041
7041	Fired Up (2009)	4.762200
8485	The Hundred-Foot Journey (2014)	4.719678
9349	Jim Jefferies: Freedumb (2016)	4.705194

```
631/631 [==============================] - 1s 2ms/step
Precision: 0.6500
Recall: 0.7791
F1-Score: 0.7087
```

Appendix E. KNN

K-Nearest Neighbour (KNN)

```
def knn_recommendation(user_item_matrix, movie_title, k=10):
    nbrs = NearestNeighbors(n_neighbors=k, metric='cosine').fit(user_item_matrix.values.T)  # fit k nearest neighbors model
    distances, indices = nbrs.kneighbors(user_item_matrix[movie_title].values.reshape(1, -1))  # find distances and indices of nearest neighbors
    return distances, indices
```

```
[157] def evaluate_knn_recommendation(user_item_matrix, movie_title, k=5):
    distances, indices = knn_recommendation(user_item_matrix, movie_title, k)  # get distances and indices of nearest neighbors and get recommendations
    recommendations = [user_item_matrix.columns[idx] for idx in indices.flatten() if idx != 0]

    actual_ratings, predicted_ratings = [], []
    for movie in recommendations:
        if movie != movie_title:
            actual_rating = user_item_matrix[movie_title].mean()
            predicted_rating = user_item_matrix[movie].mean()
            actual_ratings.append(actual_rating)
            predicted_ratings.append(predicted_rating)

    rmse, mae = calculate_metrics(actual_ratings, predicted_ratings)
    return rmse, mae
```

```python
movie_title = 'Toy Story (1995)'
distances, indices = knn_recommendation(user_item_matrix, movie_title) # get distances and indices of nearest neighbors for the movie

print("KNN Recommendations for", movie_title)
for i in range(1, len(indices.flatten())):
    print(i, user_item_matrix.columns[indices.flatten()[i]])


rmse, mae = evaluate_knn_recommendation(user_item_matrix, movie_title)
print("RMSE:", rmse)
print("MAE:", mae)

def evaluate_user_recommendation_classification_metrics(user_item_matrix, user_id, threshold=3.5, k=5):
    user_ratings, rated_movies = get_user_ratings(user_item_matrix, user_id)

    relevant_movies = user_item_matrix.loc[user_id][user_item_matrix.loc[user_id] > threshold].index.tolist()  # getting relevant movies rated above threshold

    recommended_movies = []
    for movie_title in rated_movies:
        distances, indices = knn_recommendation(user_item_matrix, movie_title, k)
        recommendations = [user_item_matrix.columns[idx] for idx in indices.flatten() if idx != 0]
        recommended_movies.extend(recommendations)

    recommended_movies = list(set(recommended_movies))

    true_positives = sum(1 for movie in recommended_movies if movie in relevant_movies)
    false_positives = len(recommended_movies) - true_positives
    false_negatives = len(relevant_movies) - true_positives

    precision = true_positives / (true_positives + false_positives) if (true_positives + false_positives) > 0 else 0
    recall = true_positives / (true_positives + false_negatives) if (true_positives + false_negatives) > 0 else 0
    f1_score = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    return {"Precision": precision, "Recall": recall, "F1-Score": f1_score}

user_id = 1
classification_metrics = evaluate_user_recommendation_classification_metrics(user_item_matrix, user_id)
print("Precision:", classification_metrics["Precision"])
print("Recall:", classification_metrics["Recall"])
print("F1-Score:", classification_metrics["F1-Score"])
```

```
KNN Recommendations for Toy Story (1995)
1 Toy Story 2 (1999)
2 Jurassic Park (1993)
3 Independence Day (a.k.a. ID4) (1996)
4 Star Wars: Episode IV - A New Hope (1977)
5 Forrest Gump (1994)
6 Lion King, The (1994)
7 Star Wars: Episode VI - Return of the Jedi (1983)
8 Mission: Impossible (1996)
9 Groundhog Day (1993)
RMSE: 0.44255653721540267
MAE: 0.36229508196721305
Precision: 0.36968576709796674
Recall: 1.0
F1-Score: 0.5398110661268556
```

Appendix F. RNN

Recurrent Neural Networks (RNN)

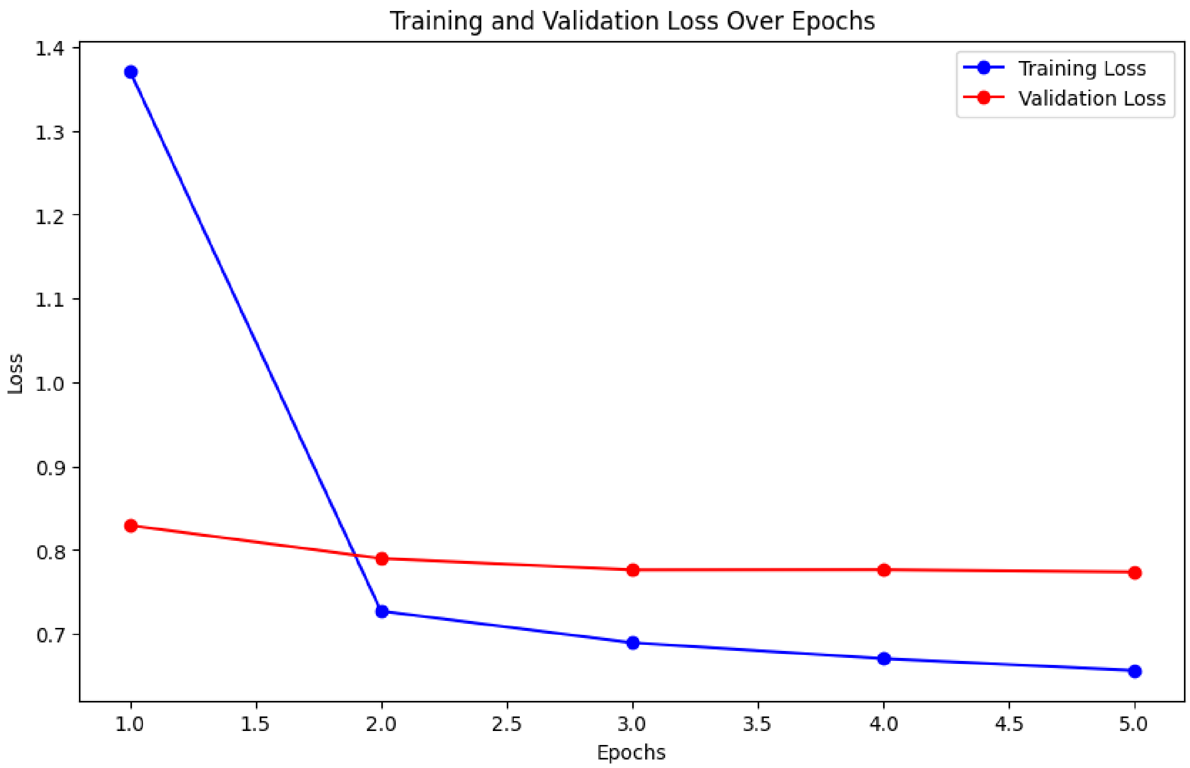
```
def prepare_sequences(data, num_users, num_movies):
    user_sequences = data['userId'].values
    movie_sequences = data['movieId'].values
    ratings = data['rating'].values
    return user_sequences, movie_sequences, ratings
```

```
[160] def create_model(num_users, num_movies):
    user_input = Input(shape=(1,))
    movie_input = Input(shape=(1,))

    user_embedding = Embedding(input_dim=num_users, output_dim=50)(user_input)
    movie_embedding = Embedding(input_dim=num_movies, output_dim=50)(movie_input)

    concatenated = Concatenate()([user_embedding, movie_embedding])
    rnn_output = SimpleRNN(units=50)(concatenated) # define SimpleRNN layer
    output = Dense(units=1, activation='linear')(rnn_output) # output layer

    model = Model(inputs=[user_input, movie_input], outputs=output) # define and compile the model
    model.compile(optimizer='adam', loss='mse')

    return model
```

```
[161] def recommend_movies(model, user_id, num_recommendations=10):
    all_movie_ids = np.arange(num_movies)
    user_ids = np.full_like(all_movie_ids, user_id)  # repeat user id for all movie ids
    predicted_ratings = model.predict([user_ids, all_movie_ids])
    top_indices = (-predicted_ratings.squeeze()).argsort()[:num_recommendations]
    recommended_movie_ids = all_movie_ids[top_indices]
    recommended_movies = movies[movies['movieId'].isin(recommended_movie_ids)]['title'].values
    return recommended_movies

encoder = LabelEncoder()
ratings['userId'] = encoder.fit_transform(ratings['userId'])
ratings['movieId'] = encoder.fit_transform(ratings['movieId'])
ratings['rating'] = ratings['rating'].astype(np.float32)

train_data, test_data = train_test_split(ratings, test_size=0.2, random_state=42) # split data

num_users = ratings['userId'].nunique()
num_movies = ratings['movieId'].nunique()

train_user_sequences, train_movie_sequences, train_ratings = prepare_sequences(train_data, num_users, num_movies) # prepare training sequences

model = create_model(num_users, num_movies) # create the recommendation model
history = model.fit(x=[train_user_sequences, train_movie_sequences], y=train_ratings, epochs=5, batch_size=64, validation_split=0.1) # train the model

user_id = 10
recommended_movies = recommend_movies(model, user_id) # get recommended movies for the specified user
print("Recommended movies for user", user_id, ":")
for i, movie in enumerate(recommended_movies, 1):
```

```
Epoch 1/5
1135/1135 [==============================] - 16s 12ms/step - loss: 1.3713 - val_loss: 0.8294
Epoch 2/5
1135/1135 [==============================] - 11s 10ms/step - loss: 0.7273 - val_loss: 0.7901
Epoch 3/5
1135/1135 [==============================] - 12s 10ms/step - loss: 0.6896 - val_loss: 0.7763
Epoch 4/5
1135/1135 [==============================] - 12s 11ms/step - loss: 0.6706 - val_loss: 0.7765
Epoch 5/5
1135/1135 [==============================] - 11s 10ms/step - loss: 0.6565 - val_loss: 0.7739
304/304 [==============================] - 1s 2ms/step
Recommended movies for user 10 :
1 . Miracle on 34th Street (1994)
2 . House Arrest (1996)
3 . Little Princess, The (1939)
4 . Hocus Pocus (1993)
5 . Rocky II (1979)
6 . Gate, The (1987)
7 . Secret Window (2004)
```

```python
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
epochs = range(1, len(training_loss) + 1)

plt.figure(figsize=(10, 6))
plt.plot(epochs, training_loss, 'bo-', label='Training Loss')
plt.plot(epochs, validation_loss, 'ro-', label='Validation Loss')
plt.title('Training and Validation Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



[163]
```python
test_user_sequences, test_movie_sequences, test_ratings = prepare_sequences(test_data, num_users, num_movies)
predicted_ratings = model.predict([test_user_sequences, test_movie_sequences]).flatten()

rmse, mae = calculate_metrics(test_ratings, predicted_ratings)
print("RMSE:", rmse)
print("MAE:", mae)
```

```
631/631 [==============================] - 1s 2ms/step
RMSE: 0.8837579036720726
MAE: 0.6767721
```

[164]
```python
classification_metrics = calculate_classification_metrics(test_ratings, predicted_ratings.squeeze(), threshold=3.5)

print("Precision:", classification_metrics["Precision"])
print("Recall:", classification_metrics["Recall"])
print("F1-Score:", classification_metrics["F1-Score"])
```

```
Precision: 0.667575179431271
Recall: 0.7582292849035187
F1-Score: 0.7100202918156343
```

Appendix G. Content-Based Filtering

Content-Based Filtering Algorithm

```
def extract_features(data):
    tfidf = TfidfVectorizer(stop_words='english') # initialize TF-IDF vectorizer
    tfidf_matrix = tfidf.fit_transform(data['genres'])  # transform genre data into TF-IDF matrix
    return tfidf_matrix

def build_similarity_matrix(data):
    feature_matrix = extract_features(data)  # extract features from movie genres
    cosine_sim_matrix = cosine_similarity(feature_matrix, feature_matrix)
    return cosine_sim_matrix

def recommend_movies_by_genre(genre, movies, top_n=10):
    genre_movies = movies[movies['genres'].str.contains(genre, case=False, na=False)] # filter movies based on the specified genre
    return genre_movies.sample(n=top_n if len(genre_movies) >= top_n else len(genre_movies))
```

```
[166] from IPython.display import display

movies['genres'] = movies['genres'].fillna('')

input_genre = 'Animation'
recommended_movies = recommend_movies_by_genre(input_genre, movies, top_n=10)

print("Recommended Movies:")
display(recommended_movies[['title', 'genres']])
```

```
Recommended Movies:
```

	title	genres
6530	Simpsons Movie, The (2007)	Animation|Comedy
4348	Laputa: Castle in the Sky (Tenkû no shiro Rapy...	Action|Adventure|Animation|Children|Fantasy|Sc...
9381	Your Name. (2016)	Animation|Drama|Fantasy|Romance
9588	Bobik Visiting Barbos (1977)	Animation|Comedy
9453	The Spirit of Christmas (1995)	Animation|Comedy
8239	Planes (2013)	Adventure|Animation|Comedy
7904	Dragon Ball: Mystical Adventure (Doragon bôru:...	Action|Adventure|Animation|Children
7903	Superman/Batman: Public Enemies (2009)	Action|Animation|Fantasy
7074	Watchmen: Tales of the Black Freighter (2009)	Action|Adventure|Animation|Horror
9334	Ice Age: Collision Course (2016)	Adventure|Animation|Children|Comedy

Appendix H. Hybrid Recommendation

Hybrid Recommendation

```python
def recommend_genre_based_movies(ratings, movies, model, user_id, genre):

    genre_movies = movies[movies['genres'].str.contains(genre, case=False, na=False)] # filtering movies by genre

    user_rated_movies = ratings[ratings['userId'] == user_id]['movieId'] # identifying movies rated by user
    unrated_movies = genre_movies[~genre_movies['movieId'].isin(user_rated_movies)]

    predictions = [(row['title'], model.predict(user_id, row['movieId']).est)
                   for _, row in unrated_movies.iterrows()]
    predictions.sort(key=lambda x: x[1], reverse=True)

    return predictions[:10]
```

```python
[243] svd_model, _ = svd_process(ratings) # get SVD model
user_id = 73
genre = 'Sci-Fi'
recommendations = recommend_genre_based_movies(ratings, movies, svd_model, user_id, genre) # get genre-based movie recommendations for the user using SVD model

recommendations_hb = pd.DataFrame(recommendations, columns=['Title', 'Predicted Rating'])

print("Recommended Movies: ")
display(recommendations_hb)
```

```python
def get_actual_and_predicted_ratings(ratings, movies, model, user_id):
    user_ratings = ratings[ratings['userId'] == user_id]
    predictions = [(row['movieId'], row['rating'], model.predict(user_id, row['movieId']).est)
                   for _, row in user_ratings.iterrows()]

    actual_ratings = [rating for _, rating, _ in predictions]
    predicted_ratings = [pred for _, _, pred in predictions]

    return actual_ratings, predicted_ratings

actual_ratings, predicted_ratings = get_actual_and_predicted_ratings(ratings, movies, svd_model, user_id)
rmse, mae = calculate_metrics(actual_ratings, predicted_ratings)
classification_metrics = calculate_classification_metrics(actual_ratings, predicted_ratings)

print(f"RMSE: {rmse:.4f}, MAE: {mae:.4f}")
print(f"Precision: {classification_metrics['Precision']:.4f}, Recall: {classification_metrics['Recall']:.4f}, F1-Score: {classification_metrics['F1-Score']:.4f}")
```

```
RMSE: 0.4949, MAE: 0.4120
Precision: 0.8171, Recall: 1.0000, F1-Score: 0.8994
```

```
Recommended Movies:
```

	Title	Predicted Rating
0	Solo (1996)	4.905276
1	Alphaville (Alphaville, une étrange aventure d...	4.750341
2	Tremors (1990)	4.649476
3	Invasion of the Body Snatchers (1978)	4.630039
4	Godzilla (1998)	4.587015
5	Star Trek II: The Wrath of Khan (1982)	4.574825
6	Making Mr. Right (1987)	4.557006
7	Children of Dune (2003)	4.551070
8	Heavy Metal 2000 (2000)	4.546609
9	Thing, The (1982)	4.534279

Appendix I. Decision Tree Classifier

Decision Tree

```
data = pd.merge(ratings, movies, on='movieId')
data['like'] = (data['rating'] > 3.5).astype(int) # create a binary 'like' column based on rating threshold (3.5)

features = data[['userId', 'movieId']] # selecting features and target variables
target = data['like']

encoder = LabelEncoder()  # encode categorical features (userId and movieId) using LabelEncoder
features.loc[:, 'userId'] = encoder.fit_transform(features['userId'])
features.loc[:, 'movieId'] = encoder.fit_transform(features['movieId'])


X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)
```

```
[171] dtree = DecisionTreeClassifier(max_depth=10, random_state=42) # train a Decision Tree Classifier

      dtree.fit(X_train, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=10, random_state=42)
```

```
[172] y_pred = dtree.predict(X_test) # making predictions on the test set
      print("Accuracy:", accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6239577129243598
```

```
[173] def recommend_movies(model, user_id, movie_data, N=10):
          unseen_movies = movie_data[~movie_data['movieId'].isin(ratings[ratings['userId'] == user_id]['movieId'])].copy()
          unseen_movies['userId'] = user_id

          unseen_movies['like_probability'] = model.predict_proba(unseen_movies[['userId', 'movieId']])[:, 1] # predict probabilities of liking for unseen movies

          recommendations = unseen_movies.sort_values('like_probability', ascending=False).head(N)
          return recommendations

      user_id = 92
      recommended_movies = recommend_movies(dtree, user_id, movies)
      print(recommended_movies[['title', 'like_probability']])
```

```
                                             title  like_probability
264                               Roommates (1995)          1.000000
551                James and the Giant Peach (1996)          1.000000
265            Ready to Wear (Pret-A-Porter) (1994)          1.000000
552                                    Fear (1996)          1.000000
553          Kids in the Hall: Brain Candy (1996)          1.000000
597                                 Thinner (1996)          1.000000
266   Three Colors: Red (Trois couleurs: Rouge) (1994)      1.000000
563  Alphaville (Alphaville, une étrange aventure d...      0.878049
562                     Run of the Country, The (1995)      0.878049
6528                                Hot Rod (2007)          0.873684
```

```
rmse = sqrt(mean_squared_error(y_test, y_pred))
mae = mean_absolute_error(y_test, y_pred)

print("RMSE:", rmse)
print("MAE:", mae)
```

```
RMSE: 0.613222869009009
MAE: 0.37604228707564025
```

```
[175] precision = precision_score(y_test, y_pred)
      recall = recall_score(y_test, y_pred)
      f1 = f1_score(y_test, y_pred)

      print("Precision:", precision)
      print("Recall:", recall)
      print("F1-Score:", f1)
```

```
Precision: 0.6297650130548302
Recall: 0.5530418832161419
F1-Score: 0.5889151135346301
```

```
[176] conf_matrix = confusion_matrix(y_test, y_pred)
      plt.figure(figsize=(8, 6))
      sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
                  xticklabels=['Not Liked', 'Liked'], yticklabels=['Not Liked', 'Liked'])
      plt.xlabel('Predicted')
      plt.ylabel('Actual')
      plt.title('Confusion Matrix')
      plt.show()
```