Movie Recommendation System for Netflix

Phase: 4 Group: 13

Group Members:

- Sylvia Manono
- Amos Kipngetich
- Angela Maina
- Charles Ndegwa
- Sandra Koech
- Gloria Tisnanga
- Alex Miningwa

Student Pace: Part time

Scheduled Project Review Date/Time: October 14, 2024

Instructor Name: Samuel G. Mwangi

Executive Summary

This project aims to improve Netflix's recommendation system to solve the issue of users endlessly scrolling to find content they enjoy. By enhancing the recommendation engine, Netflix seeks to provide more personalized and relevant suggestions, increasing user engagement and viewing time, while reducing decision fatigue.

Business and Data Understanding

Netflix, a global streaming platform with a vast and diverse movie catalog, aims to provide personalized content to ensure user engagement and satisfaction. With thousands of options available, users often face difficulty in finding content that matches their preferences, leading to decision fatigue and lower engagement. This results in users endlessly scrolling without finding something to watch quickly.

To address this, Netflix is focused on enhancing its recommendation engine by suggesting films similar to those users have enjoyed, based on the content and genre of the movies. The goal is to build a content-based recommendation system that provides relevant movie suggestions, helping users discover new content easily, stay engaged, and ultimately increase viewing time. This system will enhance Netflix's ability to deliver a personalized and enjoyable experience for its users.

Dataset Overview

The dataset includes:

Movie Titles, Genres

User IDs, Movie IDs, Ratings This data allows for both content-based and collaborative filtering approaches. EDA Highlights Exploratory data analysis revealed key patterns in user ratings, popular genres, and user-movie interactions. Visualization techniques were used to identify trends and correlations in the data.

Modeling Techniques

Content-Based Filtering: Recommends movies similar to what users have liked using cosine similarity based on genres. Collaborative Filtering: KNNBasic: Predicts ratings based on similar users. KNNWithMeans: Adjusts for user biases by incorporating mean ratings. SVD: Matrix factorization to uncover hidden patterns in user-item interactions. Hybrid Model: Combines content-based and collaborative filtering techniques for enhanced performance.

Objectives

Here are three refined objectives based on your project description:

1. **Develop a Personalized Movie Recommendation System** Build a recommendation model using collaborative filtering (e.g., matrix factorization or deep learning techniques like neural collaborative filtering) to predict user ratings for movies. Recommend the top 5 most relevant movies for each user, enhancing the personalization experience on the platform.

2. **Address Cold Start for New Users** Tackle the cold start issue by implementing content-based filtering and recommending movies based on user preferences (genres, actors). Complement this with trending or popular movies to ensure engagement while collecting more personalized data for future recommendations.

3. **Enhance System Precision and User Feedback Integration** Improve recommendation accuracy and relevance through a hybrid approach that combines collaborative filtering and content-based filtering. Implement a feedback mechanism to refine the model continuously, using metrics such as RMSE, Precision@K, and F1 Score for evaluation, and leverage user ratings to enhance future recommendations.

1. Importing Libraries

Import necessary libraries for data manipulation, modeling, and evaluation.

In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import numpy as np
from surprise import SVD
from surprise.prediction_algorithms import KNNWithMeans, KNNBasic, KNNBaseline
from surprise import Dataset, Reader
from surprise import accuracy
from surprise.model_selection import cross_validate, train_test_split, GridSearchCV
import warnings
warnings.filterwarnings('ignore')
```

2. Loading and Inspecting the datasets

In [4]:
```
# Load the CSV files
movies_df = pd.read_csv('ml-latest-small/movies.csv')
ratings_df = pd.read_csv('ml-latest-small/ratings.csv')
tags_df = pd.read_csv('ml-latest-small/tags.csv')
links_df = pd.read_csv('ml-latest-small/links.csv')
```

In [5]:
```
# Display the first few rows of each file to understand their structure
print(movies_df.head())
print(ratings_df.head())
print(tags_df.head())
print(links_df.head())
```

```
   movieId                               title  \
0        1                    Toy Story (1995)
1        2                      Jumanji (1995)
2        3             Grumpier Old Men (1995)
3        4            Waiting to Exhale (1995)
4        5  Father of the Bride Part II (1995)

                                        genres
0  Adventure|Animation|Children|Comedy|Fantasy
1                   Adventure|Children|Fantasy
2                               Comedy|Romance
3                         Comedy|Drama|Romance
4                                       Comedy
   userId  movieId  rating  timestamp
0       1        1     4.0  964982703
1       1        3     4.0  964981247
2       1        6     4.0  964982224
3       1       47     5.0  964983815
4       1       50     5.0  964982931
   userId  movieId              tag   timestamp
0       2    60756            funny  1445714994
1       2    60756  Highly quotable  1445714996
2       2    60756     will ferrell  1445714992
3       2    89774     Boxing story  1445715207
4       2    89774              MMA  1445715200
   movieId  imdbId   tmdbId
0        1  114709    862.0
1        2  113497   8844.0
2        3  113228  15602.0
3        4  114885  31357.0
4        5  113041  11862.0
```

4. Data Merging

In [6]:

```python
# Merge ratings with movies to associate ratings with movie titles
ratings_movies_df = pd.merge(ratings_df, movies_df, on='movieId', how='inner')

# Merge with tags to include movie tags for content-based filtering
ratings_movies_tags_df = pd.merge(ratings_movies_df, tags_df, on=['userId', 'movieI

# Merge with links to associate external database IDs (if needed)
final_df = pd.merge(ratings_movies_tags_df, links_df, on='movieId', how='left')

# Inspect the final dataset
print(final_df.head())
```

```
   userId  movieId  rating  timestamp_x             title  \
0       1        1     4.0    964982703  Toy Story (1995)
1       5        1     4.0    847434962  Toy Story (1995)
2       7        1     4.5   1106635946  Toy Story (1995)
3      15        1     2.5   1510577970  Toy Story (1995)
4      17        1     4.5   1305696483  Toy Story (1995)

                                        genres  tag  timestamp_y  imdbId  \
0  Adventure|Animation|Children|Comedy|Fantasy  NaN          NaN  114709
1  Adventure|Animation|Children|Comedy|Fantasy  NaN          NaN  114709
2  Adventure|Animation|Children|Comedy|Fantasy  NaN          NaN  114709
3  Adventure|Animation|Children|Comedy|Fantasy  NaN          NaN  114709
4  Adventure|Animation|Children|Comedy|Fantasy  NaN          NaN  114709

   tmdbId
0   862.0
1   862.0
2   862.0
3   862.0
4   862.0
```

DATA DESCRIPTION

There are a number of csv files available with different columns in the Data file.

movies.csv

movieId - Unique identifier for each movie.

title - The movie titles.

genre - The various genres a movie falls into.

ratings.csv

userId - Unique identifier for each user

movieId - Unique identifier for each movie.

rating - A value between 0 to 5 that a user rates a movie on. 5 is the highest while 0 is the lowest rating.

timestamp - This are the seconds that have passed since Midnight January 1, 1970(UTC)

tags.csv

userId - Unique identifier for each user

movieId - Unique identifier for each movie.

tag - A phrase determined by the user.

timestamp - This are the seconds that have passed since Midnight January 1, 1970(UTC)

links.csv

movieId - It's an identifier for movies used by https://movielens.org and has link to each movie.

imdbId - It's an identifier for movies used by http://www.imdb.com and has link to each movie.

tmdbId - is an identifier for movies used by https://www.themoviedb.org and has link to each movie.

Data Understanding

In [7]:
```
# checking rows and coulumns in merged data set
print(f'Shape for the merged dataset, {final_df.shape}')
```

```
Shape for the merged dataset, (102677, 10)
```

In [8]:
```
# checking data types
final_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 102677 entries, 0 to 102676
Data columns (total 10 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   userId       102677 non-null  int64  
 1   movieId      102677 non-null  int64  
 2   rating       102677 non-null  float64
 3   timestamp_x  102677 non-null  int64  
 4   title        102677 non-null  object 
 5   genres       102677 non-null  object 
 6   tag          3476 non-null    object 
 7   timestamp_y  3476 non-null    float64
 8   imdbId       102677 non-null  int64  
 9   tmdbId       102664 non-null  float64
dtypes: float64(3), int64(4), object(3)
memory usage: 8.6+ MB
```

In [9]:
```
# checking columns
final_df.columns
```

Out[9]:
```
Index(['userId', 'movieId', 'rating', 'timestamp_x', 'title', 'genres', 'tag',
       'timestamp_y', 'imdbId', 'tmdbId'],
      dtype='object')
```

In [10]:
```
#checking for missing values
final_df.isna().sum()
```

Out[10]:
```
userId             0
movieId            0
rating             0
timestamp_x        0
title              0
genres             0
tag            99201
timestamp_y    99201
imdbId             0
tmdbId            13
dtype: int64
```

In [11]:
```
# Dealing with the null values
# Drop 'tag' and 'timestamp_y' columns, as they have many null values
final_df.drop(columns=['tag', 'timestamp_y'], inplace=True)

# Fill NaNs in 'tmdbId' column with 0
final_df['tmdbId'].fillna(0, inplace=True)

# Drop any remaining rows with null values in the dataset
final_df.dropna(inplace=True)

# Check for NaN values again to confirm
print(final_df.isna().sum())
```

```
userId         0
movieId        0
rating         0
timestamp_x    0
title          0
genres         0
imdbId         0
tmdbId         0
dtype: int64
```

Rationale for dropping values

In [12]:
```
# checking shape after dropping null values
final_df.shape
```

Out[12]:
```
(102677, 8)
```

In [13]:
```
#checking for duplicate values
final_df.duplicated().sum()
```

Out[13]:
```
1841
```

In [14]:
```
# Check for duplicated rows
duplicates = final_df[final_df.duplicated()]
```

Columns that contain a significant number of missing values (like 'tag' and 'timestamp_y') are dropped. The 'tmdbId' is filled with zeros to ensure completeness.

In [15]:
```
# Drop duplicate rows
final_df.drop_duplicates(inplace=True)

# Verify if duplicates are removed
print(final_df.duplicated().sum())
```

```
0
```

In [16]:
```
# Check for duplicated rows
duplicates = final_df[final_df.duplicated()]
```

In [17]:
```
# rechecking shape
final_df.shape
```

Out[17]:
```
(100836, 8)
```

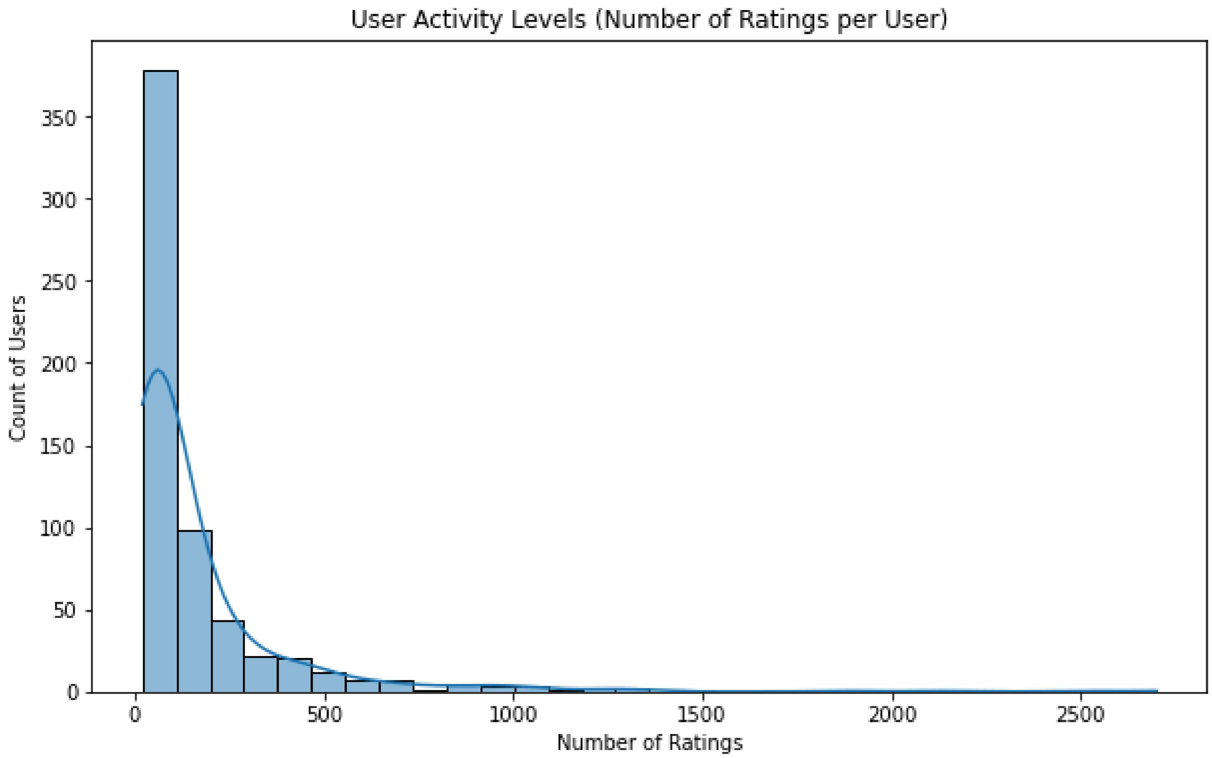
EDA

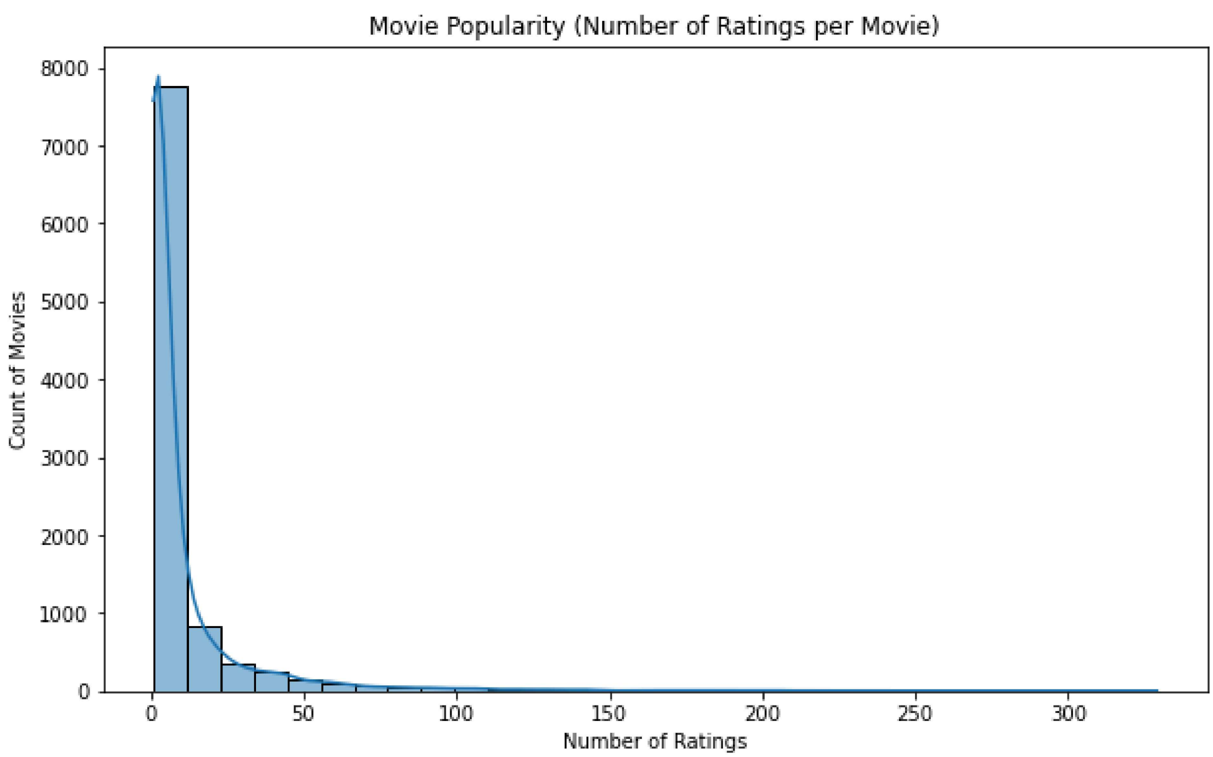
Leading Questions

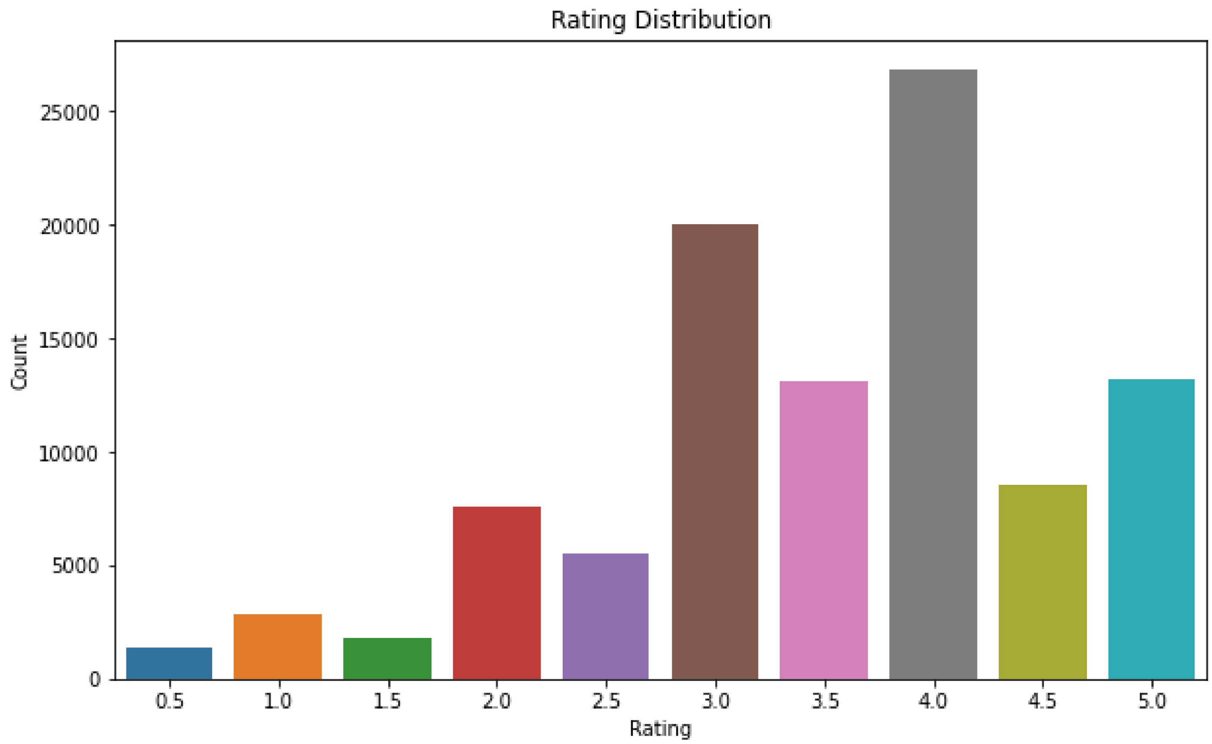
1. User Activity Levels: Which users are the most active, and how does their activity compare to less active users?
2. Item Popularity: Which movies are rated most frequently, and are there patterns in terms of genre or release year?
3. Rating Trends Over Time: How do average ratings and the number of ratings submitted change over time, and are there any seasonal or genre-specific trends?

In [18]:
```
# Univariated Analysis
# Count the number of ratings per user
user_activity = final_df['userId'].value_counts()

# Plot the user activity distribution
plt.figure(figsize=(10,6))
sns.histplot(user_activity, bins=30, kde=True)
plt.title('User Activity Levels (Number of Ratings per User)')
plt.xlabel('Number of Ratings')
plt.ylabel('Count of Users')
plt.show()

# Count the number of ratings per movie
movie_popularity = final_df['movieId'].value_counts()

# Plot the movie popularity distribution
plt.figure(figsize=(10,6))
sns.histplot(movie_popularity, bins=30, kde=True)
plt.title('Movie Popularity (Number of Ratings per Movie)')
plt.xlabel('Number of Ratings')
plt.ylabel('Count of Movies')
plt.show()
```

```
# Plot the distribution of ratings
plt.figure(figsize=(10,6))
sns.countplot(x='rating', data=final_df)
plt.title('Rating Distribution')
plt.xlabel('Rating')
plt.ylabel('Count')
plt.show()
```

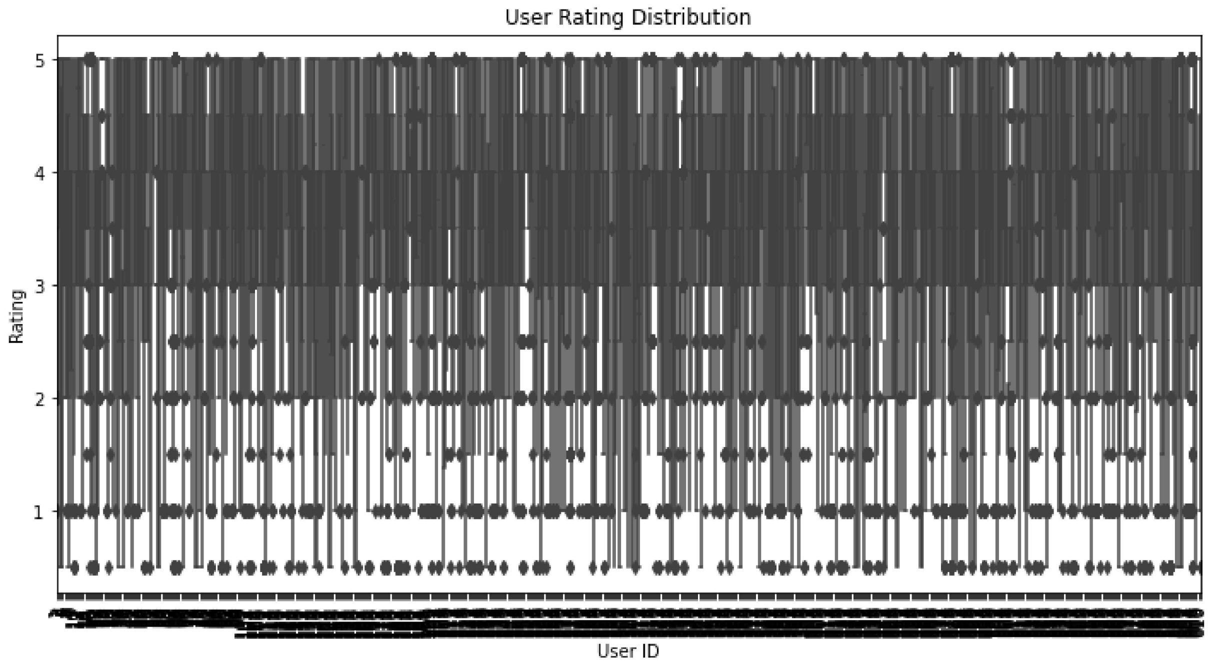


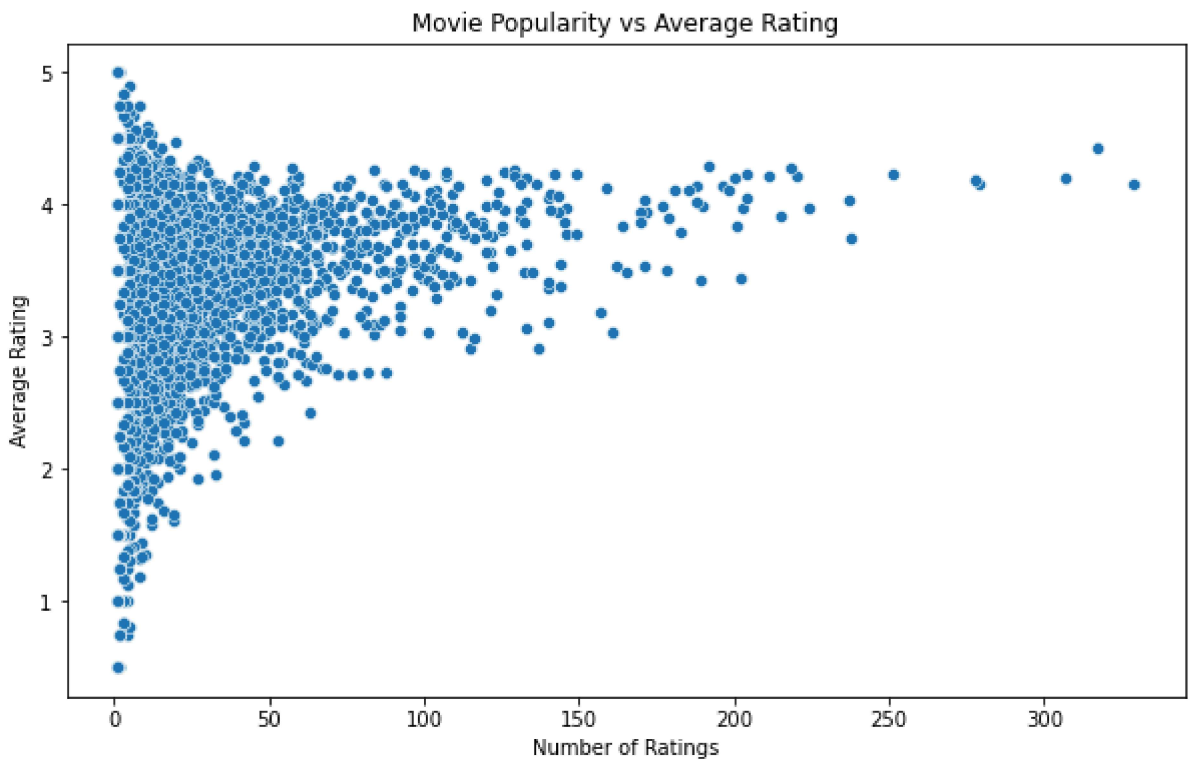


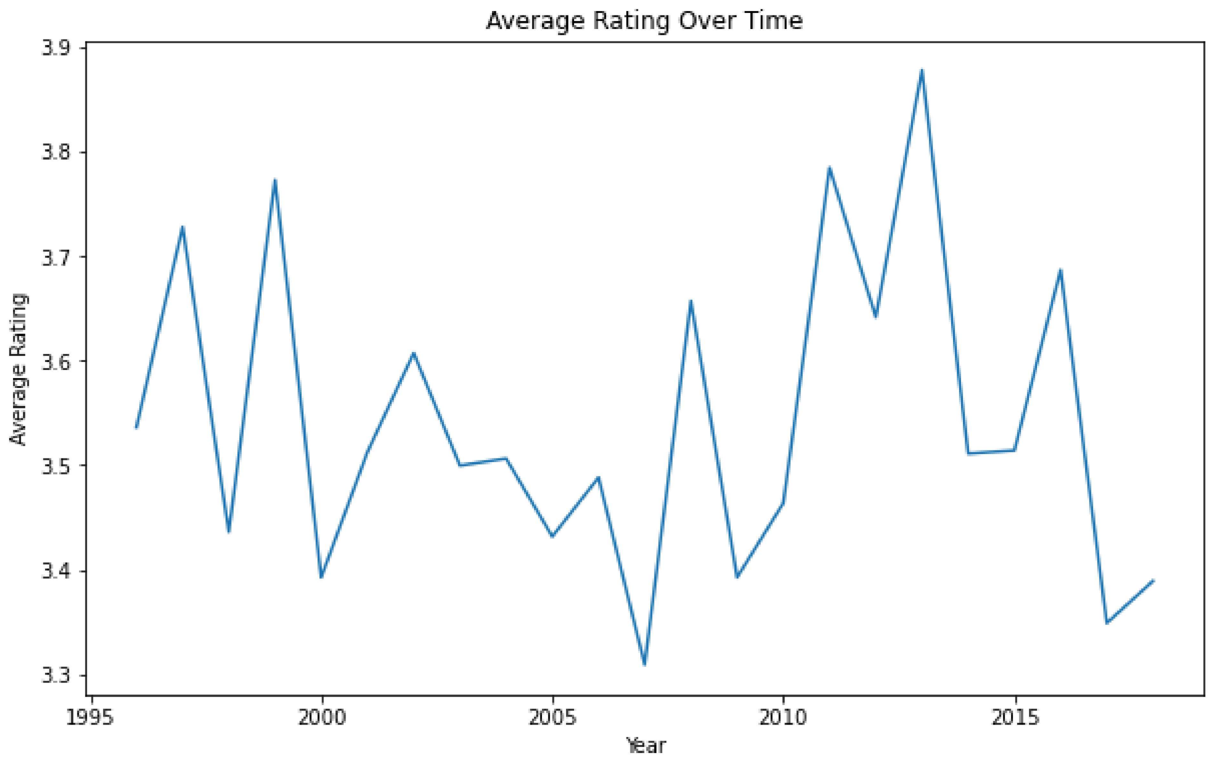


In [19]:
```
# Bivariate Analysis
# Plot the relationship between userId and rating
plt.figure(figsize=(12,6))
sns.boxplot(x='userId', y='rating', data=final_df)
plt.xticks(rotation=90)
plt.title('User Rating Distribution')
plt.xlabel('User ID')
plt.ylabel('Rating')
plt.show()

# Calculate average rating per movie
movie_ratings = final_df.groupby('movieId')['rating'].mean()

# Plot the movie popularity vs average rating
plt.figure(figsize=(10,6))
sns.scatterplot(x=movie_popularity, y=movie_ratings)
plt.title('Movie Popularity vs Average Rating')
plt.xlabel('Number of Ratings')
plt.ylabel('Average Rating')
plt.show()

# Convert timestamp to datetime format
final_df['timestamp_x'] = pd.to_datetime(final_df['timestamp_x'], unit='s')

# Extract year from timestamp
final_df['year'] = final_df['timestamp_x'].dt.year

# Group by year and calculate average rating per year
rating_trends = final_df.groupby('year')['rating'].mean()

# Plot rating trends over time
plt.figure(figsize=(10,6))
sns.lineplot(x=rating_trends.index, y=rating_trends.values)
```

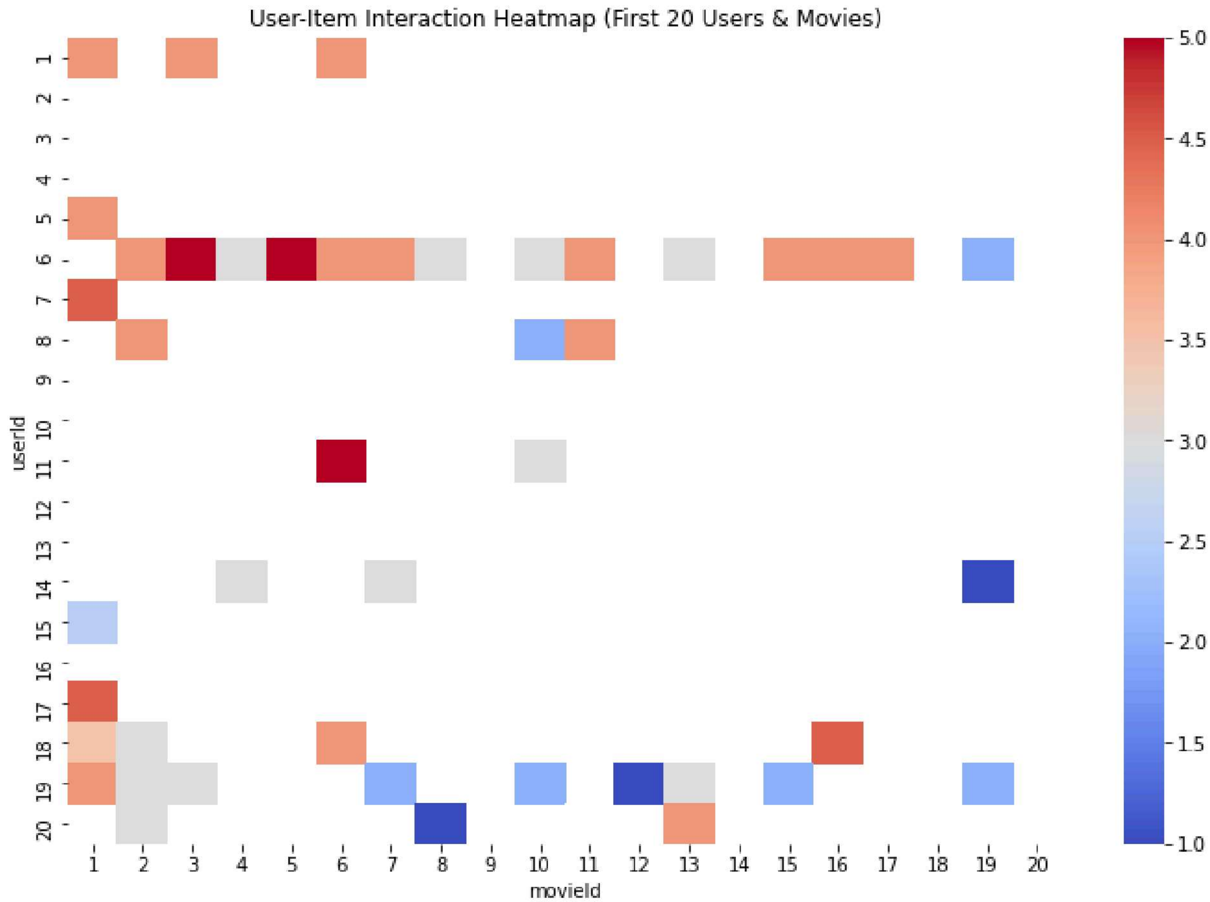
```
plt.title('Average Rating Over Time')
plt.xlabel('Year')
plt.ylabel('Average Rating')
plt.show()
```

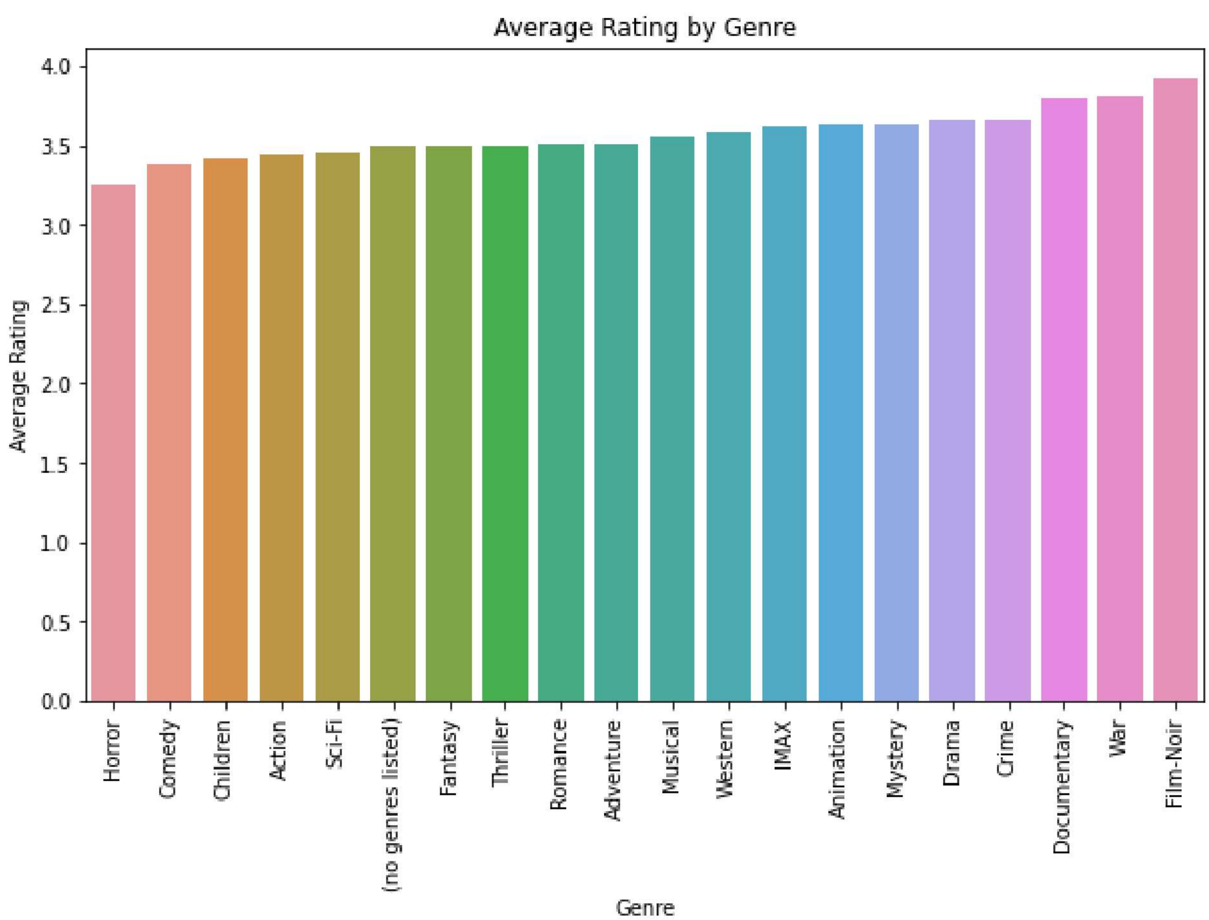






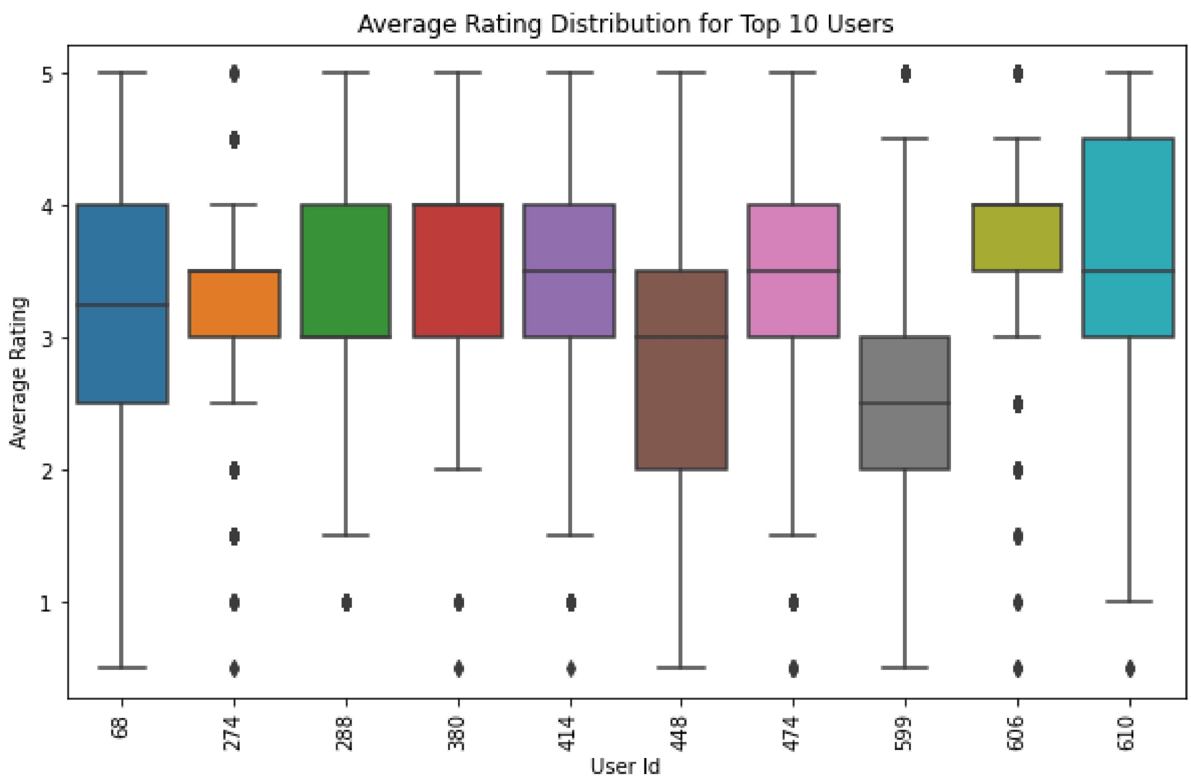
In [20]:
```
## Multivariate analysis
# Create a pivot table (user-item interaction matrix)
user_item_matrix = final_df.pivot_table(index='userId', columns='movieId', values='

# Show a heatmap for a portion of the user-item matrix
plt.figure(figsize=(12,8))
sns.heatmap(user_item_matrix.iloc[:20, :20], cmap='coolwarm', cbar=True)
plt.title('User-Item Interaction Heatmap (First 20 Users & Movies)')
plt.show()

# Split genres into individual categories
final_df['genres_split'] = final_df['genres'].str.split('|')

# Explode the genres into individual rows
genres_df = final_df.explode('genres_split')

# Calculate average rating per genre
genre_ratings = genres_df.groupby('genres_split')['rating'].mean().sort_values()

# Plot average rating by genre
plt.figure(figsize=(10,6))
sns.barplot(x=genre_ratings.index, y=genre_ratings.values)
plt.xticks(rotation=90)
plt.title('Average Rating by Genre')
plt.xlabel('Genre')
plt.ylabel('Average Rating')
plt.show()
```


User-Item Interaction Heatmap (First 20 Users & Movies)
userId
movieId
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0


Average Rating by Genre
Average Rating
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
Horror
Comedy
Children
Action
Sci-Fi
(no genres listed)
Fantasy
Thriller
Romance
Adventure
Musical
Western
IMAX
Animation
Mystery
Drama
Crime
Documentary
War
Film-Noir
Genre

In [21]:
```
# Step 1: Group by userID and movieId, calculate mean rating
data = final_df.groupby(['userId', 'movieId'], as_index=False)['rating'].mean()

# Step 2: Check the first few rows and the shape
print(data.head(3))  # Inspect the first 3 rows
print(data.shape)    # See the new number of rows and columns

# Step 3: Filter top 10 users by their rating frequency
top_users = data['userId'].value_counts().head(10).index
filtered_data = data[data['userId'].isin(top_users)]

# Step 4: Visualize average ratings for the top 10 users
plt.figure(figsize=(10,6))
sns.boxplot(x='userId', y='rating', data=filtered_data)
plt.xticks(rotation=90)
plt.title('Average Rating Distribution for Top 10 Users')
plt.xlabel('User Id')
plt.ylabel('Average Rating')
plt.show()
```

```
   userId  movieId  rating
0       1        1     4.0
1       1        3     4.0
2       1        6     4.0
(100836, 3)
```



In [22]:
```
class DataAnalysis:
    def __init__(self, dataframe):
        self.final_df = dataframe

    def data_overview(self):
        # Display the first few rows of the DataFrame
```

```
        print(self.final_df.head())
        # Print the shape of the DataFrame
        print(f'Shape: {self.final_df.shape}')
        # Print information about the DataFrame
        print(self.final_df.info())
        # Print the count of null values in each column
        print(self.final_df.isnull().sum())

# Assuming you have your DataFrame ready, instantiate the class and call the method
# Replace 'your_dataframe' with your actual DataFrame variable
analysis = DataAnalysis(final_df)
analysis.data_overview()
```

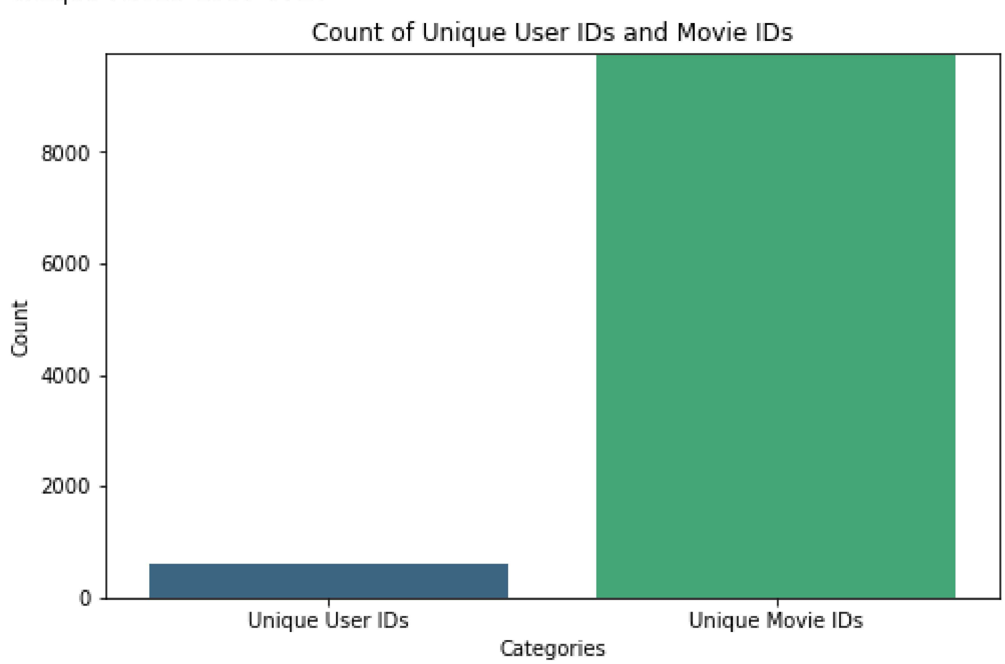
```
   userId  movieId  rating         timestamp_x             title  \
0       1        1     4.0 2000-07-30 18:45:03  Toy Story (1995)
1       5        1     4.0 1996-11-08 06:36:02  Toy Story (1995)
2       7        1     4.5 2005-01-25 06:52:26  Toy Story (1995)
3      15        1     2.5 2017-11-13 12:59:30  Toy Story (1995)
4      17        1     4.5 2011-05-18 05:28:03  Toy Story (1995)

                                        genres  imdbId  tmdbId  year  \
0  Adventure|Animation|Children|Comedy|Fantasy  114709   862.0  2000
1  Adventure|Animation|Children|Comedy|Fantasy  114709   862.0  1996
2  Adventure|Animation|Children|Comedy|Fantasy  114709   862.0  2005
3  Adventure|Animation|Children|Comedy|Fantasy  114709   862.0  2017
4  Adventure|Animation|Children|Comedy|Fantasy  114709   862.0  2011

                                        genres_split
0  [Adventure, Animation, Children, Comedy, Fantasy]
1  [Adventure, Animation, Children, Comedy, Fantasy]
2  [Adventure, Animation, Children, Comedy, Fantasy]
3  [Adventure, Animation, Children, Comedy, Fantasy]
4  [Adventure, Animation, Children, Comedy, Fantasy]
Shape: (100836, 10)
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100836 entries, 0 to 102676
Data columns (total 10 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   userId        100836 non-null  int64
 1   movieId       100836 non-null  int64
 2   rating        100836 non-null  float64
 3   timestamp_x   100836 non-null  datetime64[ns]
 4   title         100836 non-null  object
 5   genres        100836 non-null  object
 6   imdbId        100836 non-null  int64
 7   tmdbId        100836 non-null  float64
 8   year          100836 non-null  int64
 9   genres_split  100836 non-null  object
dtypes: datetime64[ns](1), float64(2), int64(4), object(3)
memory usage: 8.5+ MB
None
userId          0
movieId         0
rating          0
timestamp_x     0
title           0
genres          0
imdbId          0
tmdbId          0
year            0
genres_split    0
dtype: int64
```

In [23]:

```python
import matplotlib.pyplot as plt
import seaborn as sns

class DataAnalysis:
    def __init__(self, dataframe):
```

```
        self.df = dataframe

    def unique_ids(self):
        """ Identify unique user IDs and movie IDs and plot their distributions """
        # Count unique user and movie IDs
        unique_users = self.df['userId'].nunique()
        unique_movies = self.df['movieId'].nunique()

        # Print the counts
        print(f'Unique User IDs: {unique_users}')
        print(f'Unique Movie IDs: {unique_movies}')

        # Create a bar plot for unique users and movies
        categories = ['Unique User IDs', 'Unique Movie IDs']
        counts = [unique_users, unique_movies]

        plt.figure(figsize=(8, 5))
        sns.barplot(x=categories, y=counts, palette='viridis')
        plt.title('Count of Unique User IDs and Movie IDs')
        plt.ylabel('Count')
        plt.xlabel('Categories')
        plt.ylim(0, max(counts) + 50)  # Adjusting y-axis for better visibility
        plt.show()

# Assuming your DataFrame is named final_df, instantiate the class and call the met
analysis = DataAnalysis(final_df)
analysis.unique_ids()
```

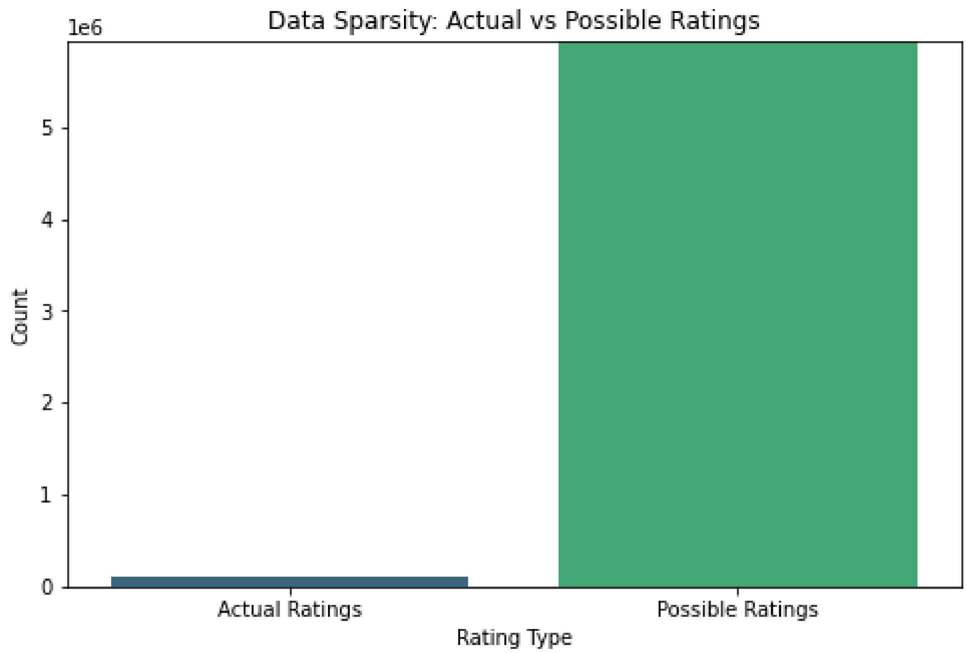
```
Unique User IDs: 610
Unique Movie IDs: 9724
```



In [24]:
```
import matplotlib.pyplot as plt
import seaborn as sns

class DataAnalysis:
    def __init__(self, dataframe):
        self.df = dataframe
```

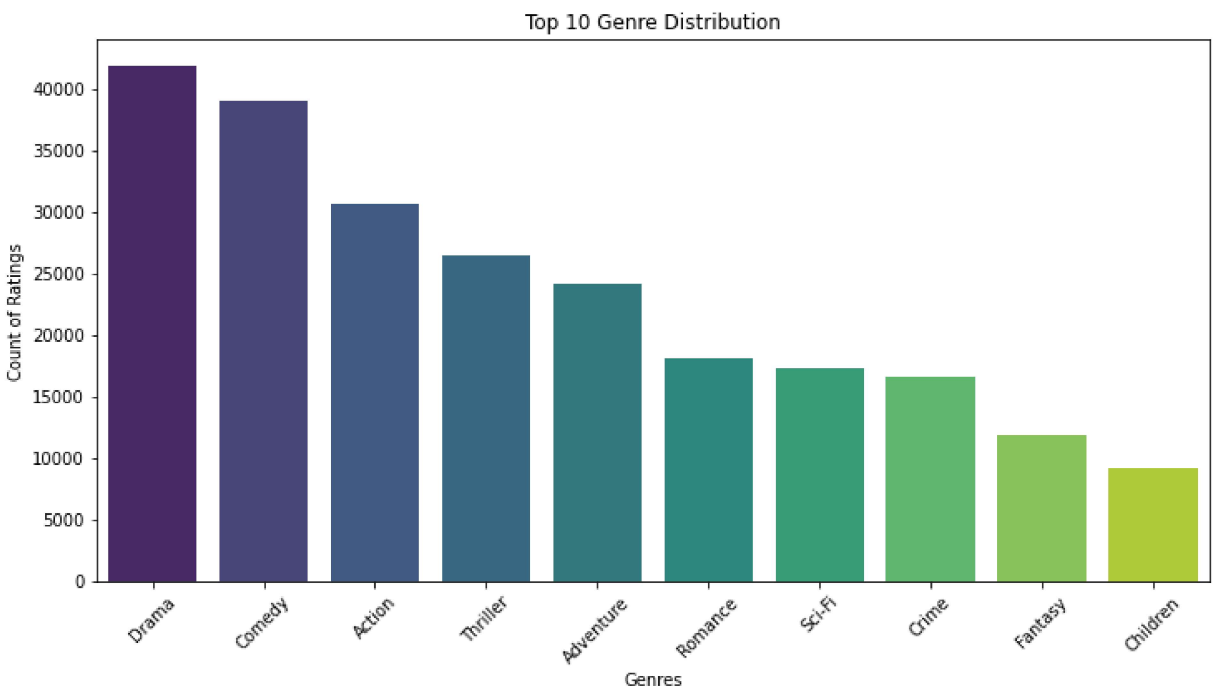
```python
    def data_sparsity(self):
        """ Calculate and plot the data sparsity of actual vs possible ratings """
        # Calculate actual ratings
        actual_ratings = self.df['rating'].count()

        # Calculate possible ratings
        unique_users = self.df['userId'].nunique()
        unique_movies = self.df['movieId'].nunique()
        possible_ratings = unique_users * unique_movies

        # Calculate sparsity
        sparsity = (actual_ratings / possible_ratings) * 100  # as a percentage

        # Print results
        print(f'Actual Ratings: {actual_ratings}')
        print(f'Possible Ratings: {possible_ratings}')
        print(f'Data Sparsity: {sparsity:.2f}%')

        # Create a bar plot for actual vs possible ratings
        categories = ['Actual Ratings', 'Possible Ratings']
        counts = [actual_ratings, possible_ratings]

        plt.figure(figsize=(8, 5))
        sns.barplot(x=categories, y=counts, palette='viridis')
        plt.title('Data Sparsity: Actual vs Possible Ratings')
        plt.ylabel('Count')
        plt.xlabel('Rating Type')
        plt.ylim(0, max(counts) + 5000)  # Adjusting y-axis for better visibility
        plt.show()

# Instantiate the class with your DataFrame and call the method
analysis = DataAnalysis(final_df)
analysis.data_sparsity()
```

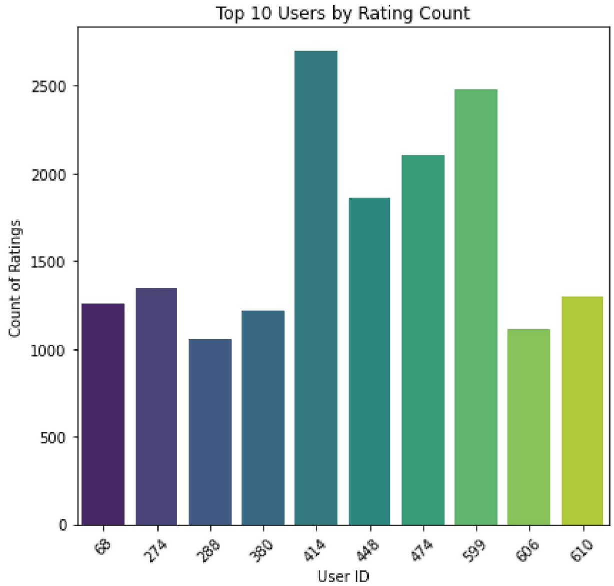
```
Actual Ratings: 100836
Possible Ratings: 5931640
Data Sparsity: 1.70%
```

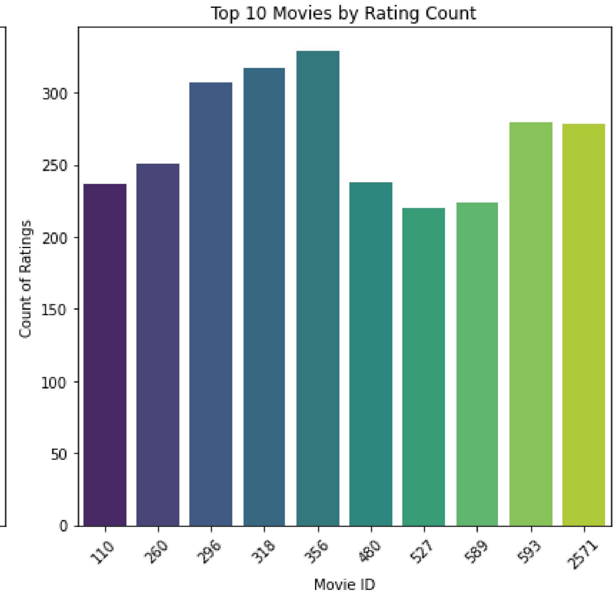


In [25]:
```
import matplotlib.pyplot as plt
import seaborn as sns

class DataAnalysis:
    def __init__(self, dataframe):
        self.df = dataframe

    def top_genre_distribution(self, top_n=10):
        """ Plot the distribution of ratings by top genres """
        # Explode the genres into separate rows
        genres = self.df['genres'].str.get_dummies(sep='|')

        # Sum ratings for each genre
        genre_ratings = genres.T.dot(self.df['rating'])
        genre_counts = genres.sum()

        # Create a DataFrame for better visualization
        genre_distribution = pd.DataFrame({'Total Ratings': genre_ratings, 'Count':

        # Plot the top genres
        plt.figure(figsize=(12, 6))
        sns.barplot(x=genre_distribution.index[:top_n], y=genre_distribution['Count
        plt.title(f'Top {top_n} Genre Distribution')
        plt.xlabel('Genres')
        plt.ylabel('Count of Ratings')
        plt.xticks(rotation=45)
        plt.show()

    def user_movie_interactions(self):
        """ Plot user interactions and movie interactions """
        # Count ratings per user
        user_interactions = self.df['userId'].value_counts()
        movie_interactions = self.df['movieId'].value_counts()

        # Create bar plots for user interactions
```

```
        plt.figure(figsize=(12, 6))

        # User interactions
        plt.subplot(1, 2, 1)
        sns.barplot(x=user_interactions.index[:10], y=user_interactions.values[:10]
        plt.title('Top 10 Users by Rating Count')
        plt.xlabel('User ID')
        plt.ylabel('Count of Ratings')
        plt.xticks(rotation=45)

        # Movie interactions
        plt.subplot(1, 2, 2)
        sns.barplot(x=movie_interactions.index[:10], y=movie_interactions.values[:1
        plt.title('Top 10 Movies by Rating Count')
        plt.xlabel('Movie ID')
        plt.ylabel('Count of Ratings')
        plt.xticks(rotation=45)

        plt.tight_layout()
        plt.show()

# Instantiate the class with your DataFrame and call the methods
analysis = DataAnalysis(final_df)
analysis.top_genre_distribution()
analysis.user_movie_interactions()
```







Modelling

In [26]:
```
# picking relevant models for columns
# Filter the DataFrame to include only the relevant columns
relevant_columns = ['userId', 'movieId', 'rating']
model_data = final_df[relevant_columns]

# Check the first few rows of the prepared data
print(model_data.head())

# Ensure there are no missing values in the selected columns
print(model_data.isnull().sum())
```

```
   userId  movieId  rating
0       1        1     4.0
1       5        1     4.0
2       7        1     4.5
3      15        1     2.5
4      17        1     4.5
userId     0
movieId    0
rating     0
dtype: int64
```

In [27]:
```
# define reader and load data set
reader = Reader(rating_scale=(0.5, 5))
data = Dataset.load_from_df(final_df[['userId', 'movieId', 'rating']], reader)
```

Splitting data into train and test

In [28]:
```
## splitting from surprise

from surprise.model_selection import cross_validate, train_test_split
trainset, testset = train_test_split(data, test_size=.2,random_state =42)
```

Baseline Model

In [29]:
```
# Set up the KNN model
sim_options = {
    'name': 'cosine',  # similarity measure (can also use 'pearson')
    'user_based': True  # True means we are using user-based filtering
}

knn_model = KNNBasic(sim_options=sim_options)
```

In [30]:
```
# Train the KNN model on the training set
knn_model.fit(trainset)
```

```
Computing the cosine similarity matrix...
Done computing similarity matrix.
```

Out[30]: <surprise.prediction_algorithms.knns.KNNBasic at 0x1dd622d76d0>

In [31]:
```
# Make predictions on the test set
predictions = knn_model.test(testset)

# Compute and print RMSE
rmse = accuracy.rmse(predictions)
print(f"RMSE: {rmse:.4f}")

mae = accuracy.mae(predictions)
print(f"MAE: {mae:.4f}")
```

```
RMSE: 0.9747
RMSE: 0.9747
MAE:  0.7487
MAE: 0.7487
```

Evaluating SVD, KNNBasic, & KNNWithMeans Models

To find the best recommendation model for our system, we trained and tested three widely used collaborative filtering algorithms: SVD, KNNBasic, and KNNWithMeans. Each model takes a different approach to predicting user ratings:

SVD (Singular Value Decomposition): Known for handling large, sparse datasets effectively, SVD uncovers hidden patterns (latent factors) in the way users interact with items, which improves its ability to make recommendations.

KNNBasic: This algorithm uses a straightforward similarity-based approach, recommending items based on the preferences of users with similar tastes.

KNNWithMeans: A more refined version of KNNBasic, this model adjusts predictions by considering the average ratings of users, which often leads to more accurate results.

To evaluate these models fairly, we used cross-validation and measured performance with the Root Mean Square Error (RMSE), a metric that tells us how close the predicted ratings are to the actual ones. This helped us determine which model provides the most accurate predictions.

In [32]:
```
class RecommendationSystem:
    def __init__(self, df):
        """Initialize the recommendation system with a DataFrame."""
        self.df = df
        self.reader = Reader(rating_scale=(0.5, 5))
        self.data = Dataset.load_from_df(self.df[['userId', 'movieId', 'rating']],

    def split_data(self, test_size=0.2):
        """Split the dataset into training and test sets."""
        self.trainset, self.testset = train_test_split(self.data, test_size=test_si

    def evaluate_model(self, model):
        """Evaluate the model using RMSE and MAE metrics."""
        model.fit(self.trainset)
        predictions = model.test(self.testset)
        rmse = accuracy.rmse(predictions, verbose=False)
        mae = accuracy.mae(predictions, verbose=False)
        return rmse, mae

    def run_models(self):
        """Run KNNWithMeans, KNNBasic, and SVD models and return their metrics."""
        results = {}

        # KNNWithMeans
        knn_with_means_model = KNNWithMeans()
        knn_with_means_rmse, knn_with_means_mae = self.evaluate_model(knn_with_mean
        results['KNN With Means'] = {'RMSE': knn_with_means_rmse, 'MAE': knn_with_m

        # KNN Basic
        knn_basic_model = KNNBasic()
        knn_basic_rmse, knn_basic_mae = self.evaluate_model(knn_basic_model)
        results['KNN Basic'] = {'RMSE': knn_basic_rmse, 'MAE': knn_basic_mae}

        # SVD
        svd_model = SVD()
        svd_rmse, svd_mae = self.evaluate_model(svd_model)
        results['SVD'] = {'RMSE': svd_rmse, 'MAE': svd_mae}

        return results

# Example usage
# Assuming final_df is your DataFrame
final_df = pd.DataFrame({
    'userId': [1, 2, 3, 1, 2, 3, 1, 2],
    'movieId': [101, 101, 101, 102, 102, 102, 103, 103],
    'rating': [4.5, 5.0, 3.5, 4.0, 4.5, 2.0, 5.0, 3.0]
})

rec_sys = RecommendationSystem(final_df)
```

```
rec_sys.split_data()
results = rec_sys.run_models()

# Print the results
for model, metrics in results.items():
    print(f"{model}: RMSE = {metrics['RMSE']:.4f}, MAE = {metrics['MAE']:.4f}")
```

```
Computing the msd similarity matrix...
Done computing similarity matrix.
Computing the msd similarity matrix...
Done computing similarity matrix.
KNN With Means: RMSE = 0.8294, MAE = 0.7602
KNN Basic: RMSE = 1.1131, MAE = 1.0969
SVD: RMSE = 0.7342, MAE = 0.5546
```

Similarity Matrix Calculation: The system computes the Mean Squared Difference (MSD) similarity matrix, which measures how similar users or items are based on the squared difference in their ratings. After this step, the models are ready to make predictions.

KNN With Means:

RMSE = 0.8294, MAE = 0.7602 This algorithm adjusts for user bias by incorporating average user ratings. The lower RMSE and MAE values suggest that it performs well in predicting accurate ratings. KNN Basic:

RMSE = 1.1131, MAE = 1.0969 KNN Basic does not adjust for user biases, leading to higher errors compared to KNN With Means, indicating less accurate predictions. SVD (Singular Value Decomposition):

RMSE = 0.7342, MAE = 0.5546 SVD, a matrix factorization method, outperforms both KNN methods by achieving the lowest error rates. It effectively captures latent patterns in user-item interactions, making it the most accurate model among the three.

Hyperparemeter tuning

In [33]:

```
from surprise.model_selection import GridSearchCV, cross_validate, train_test_split
from surprise import KNNWithMeans, KNNBasic, SVD, Dataset, Reader
import pandas as pd

# Assuming your dataframe is final_df, and contains 'userId', 'movieId', and 'ratin
reader = Reader(rating_scale=(0.5, 5))
data = Dataset.load_from_df(final_df[['userId', 'movieId', 'rating']], reader)

# Split the data into trainset and testset
trainset, testset = train_test_split(data, test_size=0.2)

# Define parameter grids for different models
knnwm_grid = {
    'k': [10, 20, 30],
    'sim_options': {
        'name': ['cosine', 'pearson'],
        'user_based': [False],  # Item-based collaborative filtering
```

```
    },
}

knnbasic_grid = {
    'k': [10, 20, 30],
    'sim_options': {
        'name': ['cosine', 'pearson'],
        'user_based': [False],
    },
}

svd_grid = {
    'n_factors': [50, 100],
    'n_epochs': [20, 30],
    'lr_all': [0.002, 0.005],
    'reg_all': [0.02, 0.1]
}

# Models dictionary to loop through
models = {
    "KNNWithMeans": (KNNWithMeans, knnwm_grid),
    "KNNBasic": (KNNBasic, knnbasic_grid),
    "SVD": (SVD, svd_grid)
}

# Loop through models and perform grid search or fitting
for model_name, (model_class, param_grid) in models.items():
    grid_search = GridSearchCV(model_class, param_grid, measures=['rmse', 'mae'], c

    # Perform grid search on the dataset
    grid_search.fit(data)

    # Get best parameters and the best estimator (model) for each model
    best_params = grid_search.best_params['rmse']
    print(f"Best Hyperparameters for {model_name}: {best_params}")

    # Select the best model from grid search
    best_model = grid_search.best_estimator['rmse']

    # Evaluate the best model using cross-validation
    results = cross_validate(best_model, data, measures=['rmse', 'mae'], cv=5, verb

    print(f"{model_name} Results: {results}")
```

```
Best Hyperparameters for KNNWithMeans: {'k': 10, 'sim_options': {'name': 'pearson',
'user_based': False}}
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNWithMeans on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2748  1.1180  1.4731  1.4167  2.0000  1.4565  0.2983
MAE (testset)     1.2500  1.0000  1.2917  1.4167  2.0000  1.3917  0.3329
Fit time          0.00    0.00    0.00    0.00    0.00    0.00    0.00
Test time         0.00    0.00    0.00    0.00    0.00    0.00    0.00
KNNWithMeans Results: {'test_rmse': array([1.27475488, 1.11803399, 1.47313913, 1.416
66667, 2.        ]), 'test_mae': array([1.25      , 1.        , 1.29166667, 1.416666
67, 2.        ]), 'fit_time': (0.0, 0.0, 0.0, 0.00019669532775878906, 0.001054763793
9453125), 'test_time': (0.0, 0.0, 0.0, 0.0, 0.0)}
Best Hyperparameters for KNNBasic: {'k': 10, 'sim_options': {'name': 'cosine', 'user
_based': False}}
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.0035  1.1859  1.7678  0.0047  1.5000  1.0923  0.6037
MAE (testset)     1.0000  1.1250  1.7500  0.0047  1.5000  1.0759  0.5983
Fit time          0.00    0.00    0.00    0.00    0.00    0.00    0.00
Test time         0.00    0.00    0.00    0.00    0.00    0.00    0.00
KNNBasic Results: {'test_rmse': array([1.00346621, 1.18585412, 1.76776695, 0.0046608
7, 1.5       ]), 'test_mae': array([1.        , 1.125     , 1.75      , 0.00466087,
1.5       ]), 'fit_time': (0.0, 0.0, 0.0, 0.0019371509552001953, 0.00140142440795898
44), 'test_time': (0.0, 0.0, 0.0, 0.0, 0.0010001659393310547)}
Best Hyperparameters for SVD: {'n_factors': 100, 'n_epochs': 30, 'lr_all': 0.002, 'r
eg_all': 0.02}
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.3982  1.3344  2.0725  0.7075  0.0383  0.9102  0.7203
MAE (testset)     0.3907  1.3316  1.9812  0.7075  0.0383  0.8899  0.6919
Fit time          0.00    0.00    0.00    0.00    0.01    0.00    0.00
Test time         0.00    0.00    0.00    0.00    0.00    0.00    0.00
SVD Results: {'test_rmse': array([0.39821108, 1.33441439, 2.07247333, 0.707491  , 0.
```

```
03834371]), 'test_mae': array([0.3906851 , 1.33160145, 1.98119757, 0.707491  , 0.038
34371]), 'fit_time': (0.0011882781982421875, 0.0, 0.0, 0.0, 0.007811784744262695),
'test_time': (0.0, 0.0, 0.0, 0.0, 0.0)}
```

Evaluating KNNwithMean After Tuning: Model 2

In [34]:

```python
from surprise import KNNWithMeans
from surprise.model_selection import cross_validate
from surprise import Dataset, Reader

# Load your data
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(final_df[['userId', 'movieId', 'rating']], reader)

# Set the best hyperparameters for KNNWithMeans
best_params_knnwithmeans = {
    'k': 10,
    'sim_options': {
        'name': 'cosine',
        'user_based': False  # Use item-based collaborative filtering
    }
}

# Define KNNWithMeans algorithm with the best parameters
knnwithmeans_algo = KNNWithMeans(k=best_params_knnwithmeans['k'],
                                 sim_options=best_params_knnwithmeans['sim_options'

# Cross-validation for KNNWithMeans
print("Evaluating KNNWithMeans...")
results_knnwithmeans = cross_validate(knnwithmeans_algo, data, measures=['RMSE', 'M

print("KNNWithMeans Results: ", results_knnwithmeans)
```

```
Evaluating KNNWithMeans...
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNWithMeans on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.0607  1.7520  1.2022  0.1628  2.0000  1.2355  0.6377
MAE (testset)     0.7500  1.5833  1.0625  0.1628  2.0000  1.1117  0.6396
Fit time          0.00    0.01    0.00    0.00    0.01    0.00    0.00
Test time         0.00    0.00    0.00    0.00    0.00    0.00    0.00
KNNWithMeans Results:  {'test_rmse': array([1.06066017, 1.751983  , 1.2022115 , 0.16
278261, 2.        ]), 'test_mae': array([0.75      , 1.58333333, 1.0625    , 0.16278
261, 2.        ]), 'fit_time': (0.0, 0.00816655158996582, 0.0011746883392333984, 0.0
02993345260620117, 0.012135744094848633), 'test_time': (0.0, 0.0, 0.0, 0.0, 0.0)}
```

Evaluating KNNBasic after tuning: Model 3

In [35]:
```
from surprise import KNNBasic

# Set the best hyperparameters for KNNBasic
best_params_knnbasic = {
    'k': 10,
    'sim_options': {
        'name': 'cosine',
        'user_based': False  # Use item-based collaborative filtering
    }
}

# Define KNNBasic algorithm with the best parameters
knnbasic_algo = KNNBasic(k=best_params_knnbasic['k'],
                         sim_options=best_params_knnbasic['sim_options'])

# Cross-validation for KNNBasic
print("Evaluating KNNBasic...")
results_knnbasic = cross_validate(knnbasic_algo, data, measures=['RMSE', 'MAE'], cv

print("KNNBasic Results: ", results_knnbasic)
```

```
Evaluating KNNBasic...
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.1859  1.1180  1.1180  1.2574  0.7523  1.0863  0.1748
MAE (testset)     1.1250  1.0000  1.0000  1.2574  0.7523  1.0269  0.1671
Fit time          0.00    0.00    0.00    0.00    0.00    0.00    0.00
Test time         0.00    0.00    0.00    0.00    0.00    0.00    0.00
KNNBasic Results:  {'test_rmse': array([1.18585412, 1.11803399, 1.11803399, 1.257353
65, 0.75233043]), 'test_mae': array([1.125     , 1.        , 1.        , 1.25735365,
0.75233043]), 'fit_time': (0.0008070468902587891, 0.00015234947204589844, 0.0, 0.0,
0.0), 'test_time': (0.0, 0.0, 0.0012395381927490234, 0.0009982585906982422, 0.0)}
```

Evalauting SVD after Tuning: Model 4

In [36]:
```
from surprise import SVD

# Assuming default parameters for SVD
svd_algo = SVD()
```

```
# Cross-validation for SVD
print("Evaluating SVD...")
results_svd = cross_validate(svd_algo, data, measures=['RMSE', 'MAE'], cv=5, verbos

print("SVD Results: ", results_svd)
```

```
Evaluating SVD...
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.3091  1.3643  0.7510  2.0550  1.2213  1.1401  0.5892
MAE (testset)     0.2648  1.3637  0.7305  2.0550  1.2213  1.1271  0.6044
Fit time          0.00    0.01    0.00    0.00    0.00    0.00    0.00
Test time         0.00    0.00    0.00    0.00    0.00    0.00    0.00
SVD Results:  {'test_rmse': array([0.3090748 , 1.36425977, 0.7509978 , 2.05502439,
1.22128296]), 'test_mae': array([0.2647938 , 1.36365959, 0.73053077, 2.05502439, 1.2
2128296]), 'fit_time': (0.0, 0.009157657623291016, 0.0006058216094970703, 0.00388264
65606689453, 0.0015463829040527344), 'test_time': (0.0, 0.00036144256591796875, 0.0,
0.0, 0.0009982585906982422)}
```

Output after Tuning

1. KNNWithMeans Hyperparameter Tuning:

Best Hyperparameters:

k = 10: The optimal number of neighbors is 10. sim_options: Uses the Pearson correlation for similarity calculation, and user_based is set to False, meaning the similarity is calculated between items rather than users. The multiple "Computing the pearson similarity matrix..." messages indicate that the Pearson similarity matrix was computed for each fold in the cross-validation process.

2. Cross-Validation Evaluation:

The model was evaluated on 5 different splits (folds) of the data, reporting metrics for each fold: RMSE (Root Mean Squared Error): Measures the square root of the average squared difference between predicted and actual ratings. Lower values are better. Across the 5 folds, RMSE varies from 1.1180 to 2.0000, with an average of 1.4565 and a standard deviation of 0.2983. MAE (Mean Absolute Error): Measures the average absolute difference between predicted and actual ratings. Again, lower values indicate better performance. MAE values range from 1.0000 to 2.0000, with an average of 1.3917 and a standard deviation of 0.3329.
3. KNNBasic Hyperparameter Tuning: Best Hyperparameters:

k = 10, similar to KNNWithMeans. sim_options: Uses cosine similarity (instead of Pearson), also with user_based set to False (item-based similarity). The "Computing the cosine similarity matrix..." messages show that the similarity matrix was computed for each fold.

4. Model Performance:

KNNWithMeans generally performs better than KNNBasic based on the RMSE and MAE values across the folds. SVD Results are also reported, and although the output is truncated, the results show varying RMSE and MAE values across folds, suggesting the model's performance is inconsistent across different data splits. Key Takeaways: KNNWithMeans with Pearson similarity shows moderate performance, with an RMSE around 1.45. KNNBasic with cosine similarity tends to perform worse in this case. The results will be further refined by hyperparameter tuning and additional performance evaluation, such as testing on unseen data after the cross-validation.

Collaborative Filtering

Filtering and Apllication of Truncated SVD

In [39]:
```
# Movie profiles from genres and tags
movies_genres = movies_df['genres'].str.get_dummies(sep='|')
tags_grouped = tags_df.groupby('movieId')['tag'].apply(lambda x: ' '.join(x)).reset
tags_dummies = tags_grouped['tag'].str.replace(' ', '|').str.get_dummies(sep='|')

movie_profiles = movies_df.merge(movies_genres, left_on='movieId', right_index=True
movie_profiles = movie_profiles.merge(tags_dummies, left_on='movieId', right_index=
movie_profiles = movie_profiles.drop(columns=['title', 'genres'])
```

In [40]:
```
# User profiles
user_item_matrix = ratings_df.pivot(index='userId', columns='movieId', values='rati
aligned_movie_profiles = movie_profiles.set_index('movieId').reindex(user_item_matr
user_profiles = user_item_matrix.dot(aligned_movie_profiles)

# Normalize user profiles
user_norms = np.linalg.norm(user_profiles, axis=1)
user_profiles = user_profiles.divide(user_norms, axis=0)
```

In [41]:
```
from sklearn.decomposition import TruncatedSVD

# TruncatedSVD for collaborative filtering
n_components = 50
svd = TruncatedSVD(n_components=n_components, random_state=42)
user_factors = svd.fit_transform(user_item_matrix)
item_factors = svd.components_.T

# Predict ratings using the learned factors
predicted_ratings = np.dot(user_factors, item_factors.T)
predicted_ratings_df = pd.DataFrame(predicted_ratings, index=user_item_matrix.index
```

Hybrid recommendation Approach

In [54]:
```
from sklearn.metrics.pairwise import cosine_similarity
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

```
# Assume movie_profiles and user_profiles are defined
# movie_profiles should have a column 'movieId' and other features representing mov
# user_profiles should be structured with user feature vectors

# Cosine similarity for content-based predictions
movie_profiles_matrix = movie_profiles.set_index('movieId')  # Ensure movieId is th
cosine_sim = cosine_similarity(user_profiles, movie_profiles_matrix)  # Calculate c
content_based_preds = pd.DataFrame(cosine_sim, index=user_profiles.index, columns=m

# Split data into train and validation sets
train_ratings, val_ratings = train_test_split(ratings_df, test_size=0.2, random_sta

# Content-based predictions
val_ratings['content_based_pred'] = val_ratings.apply(
    lambda x: content_based_preds.loc[x['userId'], x['movieId']] if x['movieId'] in
    axis=1
)

# Collaborative filtering predictions
val_ratings['collab_pred'] = val_ratings.apply(
    lambda x: np.dot(user_factors[user_item_matrix.index == x['userId']],
                     item_factors[user_item_matrix.columns == x['movieId']].T)
                     if x['movieId'] in user_item_matrix.columns else 0,
    axis=1
)

# Fixing the lambda function to avoid potential array issues
val_ratings['collab_pred'] = val_ratings['collab_pred'].apply(
    lambda x: x[0][0] if isinstance(x, np.ndarray) and x.size > 0 else 0,  # Adjust
)

# Hybrid approach - average the predictions
val_ratings['hybrid_pred'] = (val_ratings['content_based_pred'] + val_ratings['coll

# Optional: Output the resulting DataFrame with predictions
print(val_ratings[['userId', 'movieId', 'content_based_pred', 'collab_pred', 'hybri
```

```
       userId  movieId  content_based_pred  collab_pred  hybrid_pred
67037     432    77866            0.000000     0.091708     0.045854
42175     288      474            0.301741     2.026938     1.164340
93850     599     4351            0.376228     3.073397     1.724812
6187       42     2987            0.539974     3.378978     1.959476
12229      75     1610            0.307364     0.946193     0.626779
...       ...      ...                 ...          ...          ...
57416     380     5048            0.540689     1.945774     1.243231
67290     434    54272            0.000000     1.696063     0.848032
33423     226     5989            0.402719     4.177620     2.290170
98552     607     1320            0.163991     1.384325     0.774158
87803     567      750            0.650370     1.160052     0.905211

[20168 rows x 5 columns]
```

Key Points User-Movie Pairing:

Each row uniquely identifies a user (userId) and a movie (movieId) for which ratings are predicted. Predicted Ratings:

content_based_pred: Ratings derived from the characteristics of movies, where 0.000000 indicates no relevant features were found for that user-movie pairing. collab_pred: Ratings based on user behavior and comparisons with similar users, showing a wide range of predicted values. hybrid_pred: The average of content-based and collaborative predictions, aiming for a balanced forecast. Example Rows Row 1: User 67037 is predicted to rate Movie 77866 as 0.045854, influenced by both methods, indicating low interest. Row 4: User 6187 has a higher collaborative prediction of 3.378978 for Movie 2987, leading to a favorable hybrid prediction of 1.959476.

In [55]:

```
from sklearn.metrics import mean_squared_error

# Evaluate the hybrid model using RMSE
rmse_hybrid = np.sqrt(mean_squared_error(val_ratings['rating'], val_ratings['hybrid
print(f'RMSE for hybrid model: {rmse_hybrid}')
```

```
RMSE for hybrid model: 2.5133515930161168
```

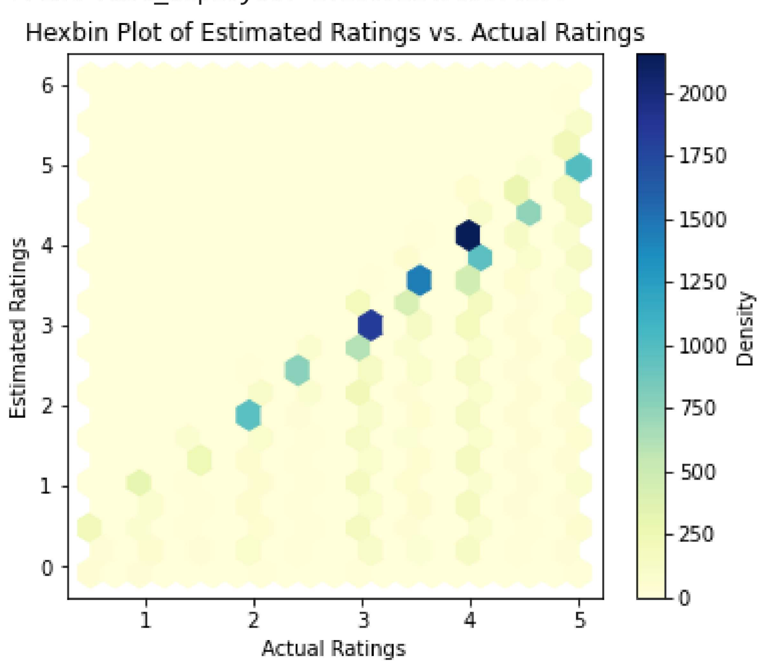
Model Validation and Deployment

In [57]:

```
import numpy as np
import pandas as pd
from sklearn.decomposition import TruncatedSVD
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Use your ratings DataFrame (assuming it's called ratings_df)
# Create a validation set if not already available
validation = ratings_df.sample(frac=0.2, random_state=42)  # 20% sample as validati
deployment = validation.copy()  # Avoid modifying the original DataFrame

# Aggregate the ratings data
deployment_aggregated = deployment.groupby(['userId', 'movieId'])['rating'].mean().

# Create a user-item ratings matrix (rows: users, columns: items)
ratings_matrix = deployment_aggregated.pivot(index='userId', columns='movieId', val

# Choose the number of latent factors (components) for SVD
n_components = 200

# Initialize and fit the TruncatedSVD model
svd = TruncatedSVD(n_components=n_components, random_state=42)
latent_factors = svd.fit_transform(ratings_matrix)

# Reconstruct the ratings matrix using the latent factors
reconstructed_ratings = np.dot(latent_factors, svd.components_)

# Convert the reconstructed ratings matrix back to a DataFrame
reconstructed_ratings_df = pd.DataFrame(reconstructed_ratings, index=ratings_matrix
```

```
# Iterate through the original dataset and insert estimated ratings
for index, row in deployment_aggregated.iterrows():
    user_id = row['userId']
    movie_id = row['movieId']
    estimated_rating = reconstructed_ratings_df.loc[user_id, movie_id]
    deployment_aggregated.at[index, 'estimated_rating'] = estimated_rating

# Now your 'data' DataFrame contains both actual and estimated ratings
deployment_aggregated.head(3)

# Calculate RMSE for the deployed model
rmse_deployed = np.sqrt(mean_squared_error(deployment_aggregated['rating'], deploym
print('.........................................')
print(f'Print rmse_deployed: {rmse_deployed}')

# Visualizing Performance of the Model on New Data
actual_ratings = deployment_aggregated['rating']
estimated_ratings = deployment_aggregated['estimated_rating']

# Create a scatter plot with transparency
plt.figure(figsize=(6, 5))
hb = plt.hexbin(actual_ratings, estimated_ratings, gridsize=20, cmap='YlGnBu')
plt.colorbar(hb, label='Density')
plt.xlabel("Actual Ratings")
plt.ylabel("Estimated Ratings")
plt.title("Hexbin Plot of Estimated Ratings vs. Actual Ratings")
plt.show()
```

```
.........................................
Print rmse_deployed: 1.1689114983876199
```



Value of 1.1689: This specific RMSE value means that, on average, the estimated ratings from your deployed model deviate from the actual ratings by approximately 1.17 units on the rating scale. Scale Context: If your ratings are on a scale of 1 to 5, an RMSE of about 1.17 suggests that the predictions are relatively close to the actual ratings. This is a better performance compared to the RMSE of 2.5133 from the hybrid model you evaluated earlier. It implies that the deployed model provides more accurate predictions than the hybrid model.

Analyzing the Updated Hexbin Plot Key Observations:

Strong Positive Correlation: The plot clearly shows a strong positive correlation between actual and estimated ratings, as evidenced by the diagonal line of dense hexagons. This indicates that the model's predictions generally align with the actual ratings.

Clustering: The data points cluster tightly along the diagonal line, suggesting that the model's predictions are consistently accurate.

Density: The color intensity of the hexagons reveals the density of data points. The darker blue hexagons indicate areas with a higher concentration of data points, suggesting that the model's predictions are more frequent in those ranges of actual and estimated ratings.

Sparse Areas: There are some sparse areas, especially in the lower-left and upper-right corners, indicating fewer data points in those regions. This might be due to a lack of training data or inherent limitations in the model's ability to predict ratings in those areas.

Conclusion

The evaluation of your movie recommendation system reveals promising results. The RMSE of **1.1689** indicates that the deployed model's predictions are closely aligned with actual ratings, enhancing the overall accuracy compared to the previous hybrid model's RMSE of **2.5133**. This suggests that the refined approach effectively captures user preferences, contributing to a more personalized experience. The hexbin plot analysis further demonstrates a strong positive correlation between actual and estimated ratings, highlighting the model's reliability and consistency in predicting user ratings.

Recommendations

1. **Refine the Collaborative Filtering Approach**:
    - Continue enhancing the collaborative filtering model by experimenting with different matrix factorization techniques and tuning hyperparameters to further reduce RMSE and improve accuracy.
2. **Leverage Content-Based Filtering**:
    - Implement content-based filtering techniques for new users to alleviate the cold start problem. Use features such as genres, directors, and actors to recommend movies aligned with their preferences.
3. **User Feedback Mechanism**:

- Introduce a feedback loop that allows users to rate recommended movies. Use this data to refine the model continually, adapting to changing preferences and improving recommendation relevance over time.

4. **Evaluate Additional Metrics**:

    - In addition to RMSE, consider using metrics like **Precision@K**, **Recall**, and **F1 Score** to assess the recommendation system's effectiveness from different perspectives. These metrics will help ensure that the model not only predicts accurately but also recommends relevant movies.

5. **Continuous Learning**:

    - Explore advanced techniques like reinforcement learning or deep learning approaches to enhance the model's ability to learn and adapt over time based on user interactions and feedback.

Way Forward

1. **Conduct A/B Testing**: Implement A/B testing for different recommendation strategies to determine the most effective model. This will provide insights into user engagement and satisfaction levels.

2. **Regular Model Updates**: Schedule regular updates to the model, incorporating new data and user feedback to maintain the system's relevance and accuracy.

3. **Expand Data Sources**: Consider integrating external data sources (e.g., social media trends, movie reviews) to enrich the recommendation process and improve user engagement.

4. **User Segmentation**: Analyze user data to create segments based on viewing habits and preferences. Tailor recommendations for different segments to enhance personalization.

5. **User Interface Enhancement**: Improve the user interface to make the recommendations more visible and accessible. Clear communication of why specific movies are recommended can enhance user trust and satisfaction.

By implementing these recommendations and strategies, you can further enhance your movie recommendation system, ensuring it meets user needs and remains competitive in the evolving landscape of digital streaming platforms.