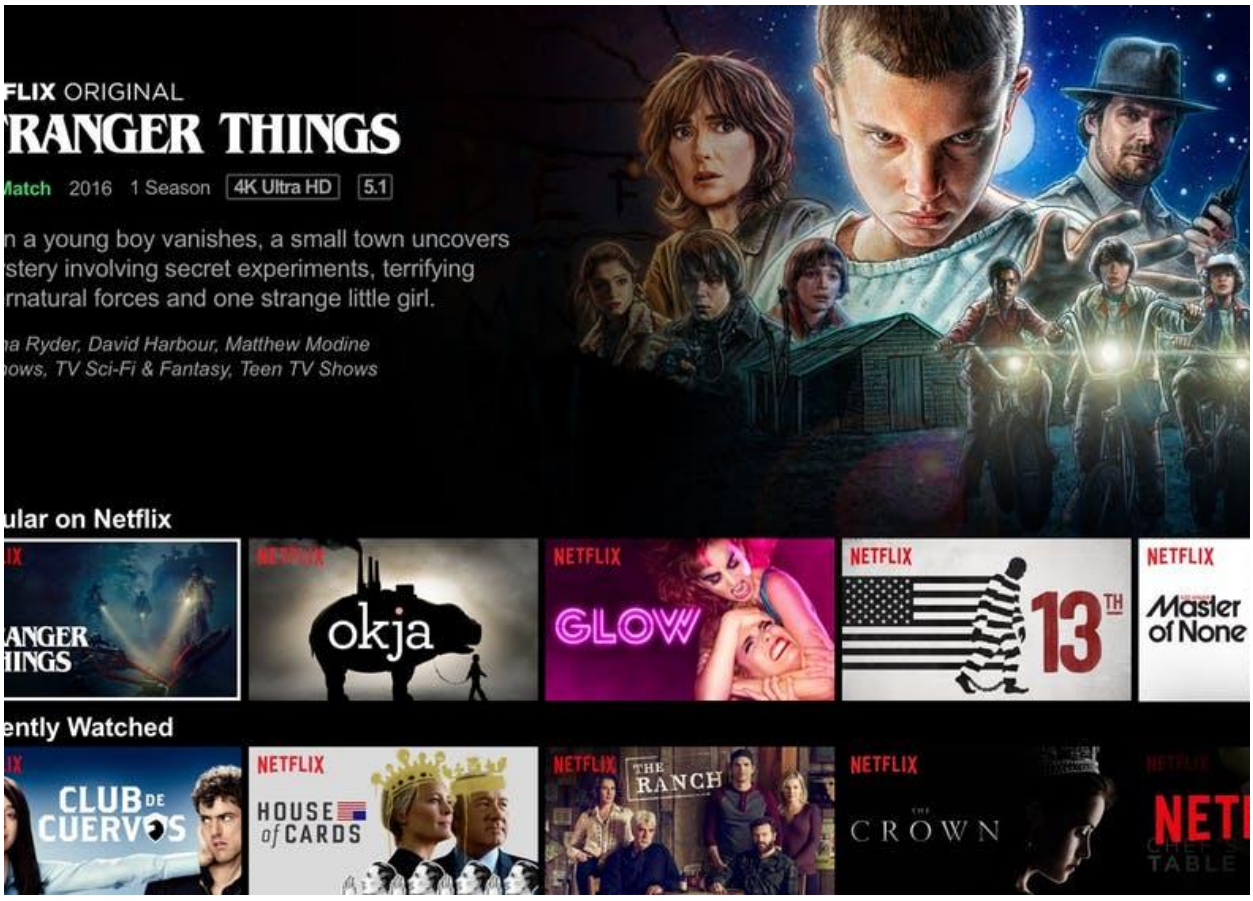
Phase 4
Recommendation System



Business Understanding

Introduction

A recommender system, or a recommendation system, is a subclass of information filtering system that seeks to predict the “rating” or “preference” a user would give to an item.A recommendation system allows predicting

the future preference list for a certain customer or user, and recommends the top preference for this user.Two most ubiquitous types of personalised recommendation systems are **Content-Based** and **Collaborative Filtering**. Collaborative filtering produces recommendations based on the knowledge of users' attitude to items, that is it uses the "wisdom of the crowd" to recommend items. In contrast, content-based recommendation systems focus on the attributes of the items and give you recommendations based on the similarity between them.

Problem Statement

Our goal is to develop a movie recommendation system that can provide personalized recommendations to users based on their ratings of other movies. By leveraging the MovieLens dataset, we aim to create a model that can accurately identify the top 5 movie recommendations for each user.

Business Objectives

Main objective

Developing a recommendation system.

Specific Objectives

1. What movie genres are most popular to the users ?
2. What genres have the most ratings ?

Performance /Evaluation Metrics

Use of RMSE to compare the error of the training and test datasets .

Data Processing

This project will involve loading and processing the links, tags, movies, and ratings data to prepare them for input into the models.

Data Overview

The dataset (ml-latest-small) describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service.

The data are contained in the files links.csv, movies.csv, ratings.csv and tags.csv.

Link to the dataset https://grouplens.org/datasets/movielens/latest/

User Ids

MovieLens users were selected at random for inclusion. User ids are consistent between ratings.csv and tags.csv (i.e., the same id refers to the same user across the two files).

Movie Ids

Only movies with at least one rating or tag are included in the dataset. Movie ids are consistent between ratings.csv, tags.csv, movies.csv, and links.csv (i.e., the same id refers to the same movie across these four data files).

Ratings Data File Structure (ratings.csv)

All ratings are contained in the file ratings.csv. Each line of this file after the header row represents one rating of one movie by one user, and has the following format:

userId,movieId,rating,timestamp

The lines within this file are ordered first by userId, then, within user, by movieId.

Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars).

Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

Tags Data File Structure (tags.csv)

All tags are contained in the file tags.csv. The lines within this file are ordered first by userId, then, within user, by movieId.

Tags are user-generated metadata about movies. Each tag is typically a single word or short phrase. The meaning, value, and purpose of a particular tag is determined by each user.

Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

Movies Data File Structure (movies.csv)

Movie information is contained in the file movies.csv.

Movie titles are entered manually or imported from https://www.themoviedb.org/, and include the year of release in parentheses. Errors and inconsistencies may exist in these titles.

Genres are a pipe-separated list, and are selected from the following:

- Action
- Adventure
- Animation
- Children's
- Comedy
- Crime
- Documentary
- Drama
- Fantasy
- Film-Noir
- Horror
- Musical
- Mystery
- Romance
- Sci-Fi
- Thriller
- War
- Western
- (no genres listed)

Data Loading & Combining

Loaded all the datasets available separately and named each as per the information contained in the dataset.

 # Reading links file

links = pd.read_csv('/content/links.csv')

 # Reading movies file

movies = pd.read_csv('/content/movies.csv')

 # Reading ratings file

ratings = pd.read_csv('/content/ratings.csv')

 # Reading tags file

tags = pd.read_csv('/content/tags.csv')

The links and tags dataset seemed to have information contained in the ratings and movies dataset ,the tags column would not be used in our project ,with that a decision to merge the movies and ratings dataset was made to come up with one dataset '***data***' .

 # merging the movies and ratings dataset to have one dataset to work with.

data = pd.merge(movies,ratings,on ='movieId')

During the cleaning proces the '*timestamp*' column was dropped from the dataset '*data*'.The columns available for modelling are now :

#confirming the drop of the column

data.info()

<class 'pandas.core.frame.DataFrame'>

Int64Index: 100836 entries, 0 to 100835

Data columns (total 5 columns):

 # Column Non-Null Count Dtype

--- ------ -------------- -----

0 movieId 100836 non-null int64

1 title 100836 non-null object

2 genres 100836 non-null object

3 userId 100836 non-null int64

4 rating 100836 non-null float64

dtypes: float64(1), int64(2), object(2)

Data Exploration

At this phase we explored and visualized our new dataset '***data***' to uncover different insights and also to identify areas or patterns to dig into.

What are the most featured genres in the Movies dataset?

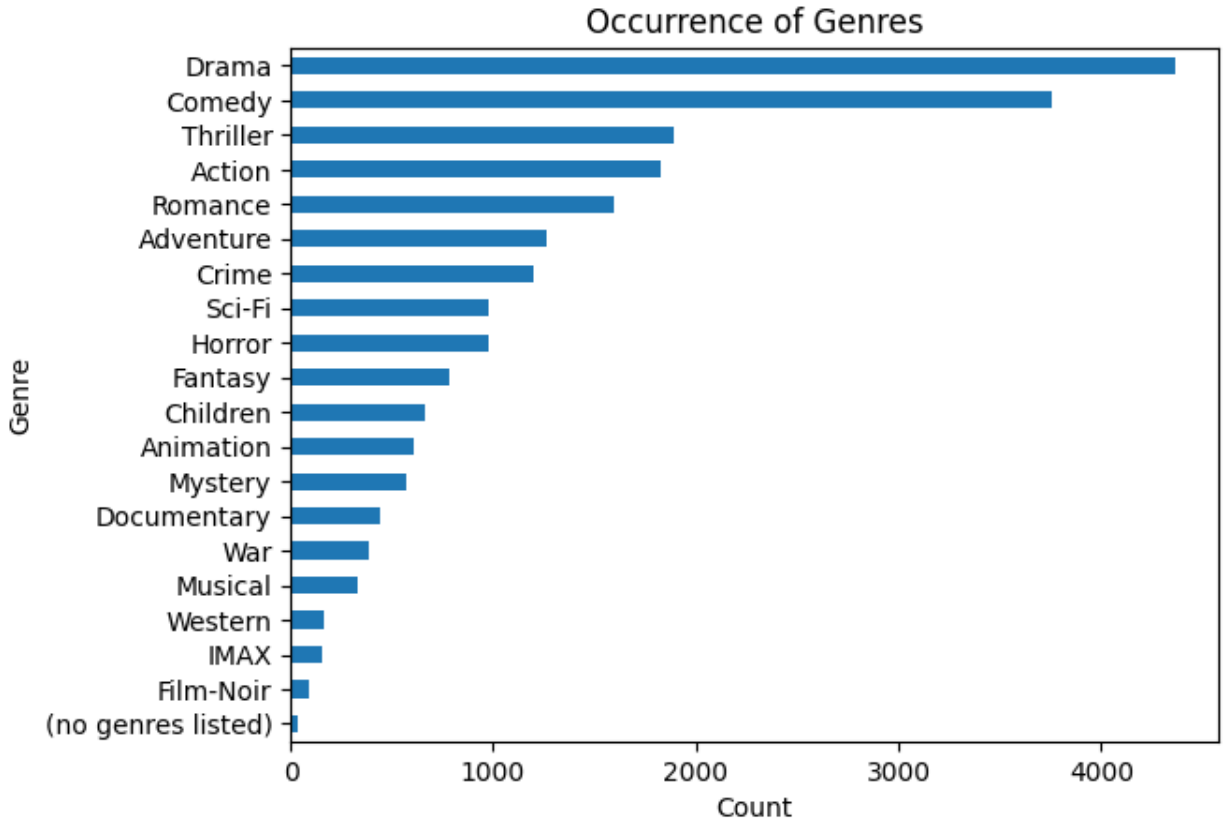
```
# Concatenate all the genre strings into a single string

all_genres = '|'.join(movies['genres'].tolist())
```

```
# Split the concatenated string into individual words
words = all_genres.split('|')

# Count the occurrence of each word
word_count = Counter(words)

# Print the occurrence of each word
genre_lst = []
count_lst = []
for word, count in word_count.items():
    genre_lst.append(word)
    count_lst.append(count)
    # genre_count[word]=count
   # Concatenate all the genre strings into a single string
all_genres = '|'.join(movies['genres'].tolist())

# Split the concatenated string into individual words
words = all_genres.split('|')

# Count the occurrence of each word
word_count = Counter(words)

# Print the occurrence of each word
```

```
genre_lst = []

count_lst = []

for word, count in word_count.items():

    genre_lst.append(word)

    count_lst.append(count)

    # genre_count[word]=count


genre_count = pd.DataFrame({"genre": genre_lst, "count":count_lst})

genre_count

genre_count = pd.DataFrame({"genre": genre_lst, "count":count_lst})

genre_count
```

Craeted a syntax that counts the occurrence of each genre and plots a bar graph against the counts

```
(genre_count

.sort_values(by=["count"], ascending=True)

.plot(x='genre', y='count', kind='barh', legend=False))


# Add labels and title

plt.xlabel('Genre')

plt.ylabel('Count')

plt.title('Occurrence of Genres')


# Display the plot

plt.show()
```



Our image above depicts the occurence of genres with our top 5 being: Drama, Comedy, Thriller, Action and Romance. We can also see our least occuring genres in our dataset being Film-Noir, IMAX, Western and Musical just to mention a few.

 # Combining all genres to one string

all_genres = ''.join(data['genres'])

wordcloud = WordCloud(width=800, height=400,
background_color='white')#.generate(all_genres)

 # creating a word cloud

wordcloud.generate(all_genres)

plt.figure(figsize=(10, 5))

```
plt.imshow(wordcloud, interpolation='bilinear')

plt.axis('off')

plt.show()
```

From the visulizations obtained above, we are able to see the popularity of the various genres our most occuring genres are drama, comedy, action, thriller and adventure etc.

What Genres have the most ratings?¶

 # Display 20 movies with highest ratings

```
top_movies = data[['title','genres','rating']].sort_values('rating',
ascending=False).head(20)

top_movies

# Create a bar plot

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

plt.barh(top_movies['title'], top_movies['rating'], color='purple')
```

plt.xlabel('Rating')

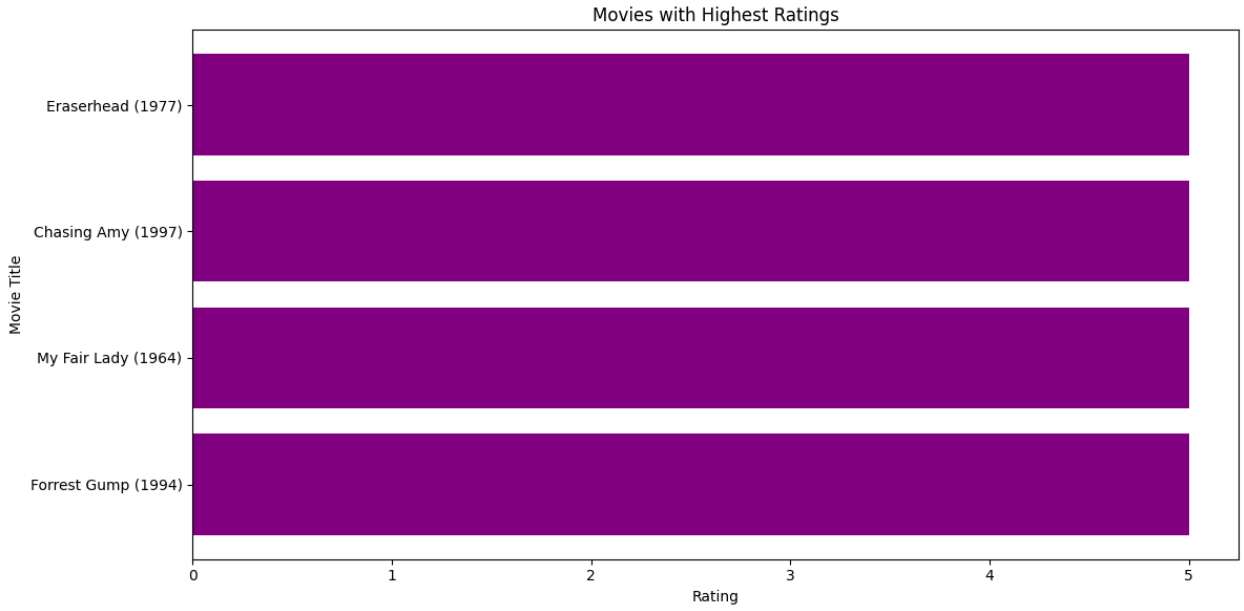
plt.ylabel('Movie Title')

plt.title('Top 20 Movies with Highest Ratings')

plt.gca().invert_yaxis() # Invert the y-axis to display the highest-rated movie at the top

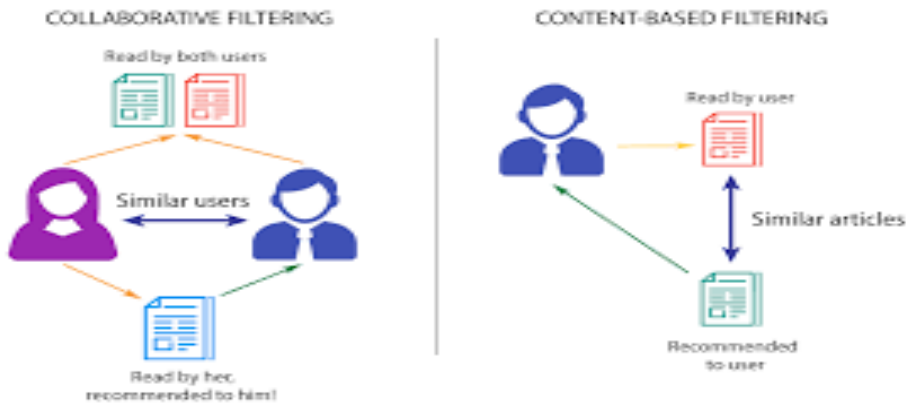
plt.tight_layout()

plt.show()



Recommendation Systems

There are two types of recommendation systems ;



1. Content-Based

The Content-Based Recommender relies on the similarity of the items being recommended. The basic idea is that if you like an item, then you will also like a “similar” item. It generally works well when it's easy to determine the context/properties of each item.

A content based recommender isworks with data that the user provides, either explicitly movie ratings for the MovieLens dataset. Based on that data, a user profile is generated, which is then used to make suggestions to the user. As the user provides more inputs or takes actions on the recommendations, the engine becomes more and more accurate.

Building a Content-Based Recommendation system that computes similarity between movies based on movie genres. It will suggest movies that are most similar to a particular movie based on its genre.

```
# Separating the genres into a string array

movies_set['genres'] = movies_set['genres'].str.split('|')

# Convert genres to string value

movies_set['genres'] = movies_set['genres'].fillna("").astype('str')
```

Using TfidfVectorizer function from scikit-learn, which transforms text to feature vectors that can be used as input to estimator.

```
from sklearn.feature_extraction.text import TfidfVectorizer



tf = TfidfVectorizer(analyzer='word',ngram_range=(1, 2),min_df=0,
stop_words='english')

tfidf_matrix = tf.fit_transform(movies_set['genres'])

tfidf_matrix.shape
```

Using the Cosine Similarity to calculate a numeric quantity that denotes the similarity between two movies.

To measure the similarity of two two genres, there are several natural distance measures we can use:

1. We could use Jaccard distance between the sets of words.
2. We could use the cosine distance between the sets treated as vectors. In our case will use the cosine similarity.

```
from sklearn.metrics.pairwise import cosine_similarity

from sklearn.feature_extraction.text import CountVectorizer
```

The movie genres are used to compute the cosine similarity matrix between all pairs of movies. This means that the recommendations are based on how similar the genres of two movies are.

```
# Create a bag of words representation of the movie genres

vectorizer = CountVectorizer(token_pattern='(?u)\\b\\w+\\b')

genres_bow = vectorizer.fit_transform(movies['genres'])

genres_bow

# Compute the cosine similarity matrix between all pairs of movies based on their genres

cosine_sim = cosine_similarity(genres_bow)

cosine_sim

movie_id = 3

movie_indices = pd.Series(movies.index, index=movies['movieId'])

similarity_scores = list(enumerate(cosine_sim[movie_indices[movie_id]]))
```

This code computes the cosine similarity scores between the movie with id 3 and all other movies in the dataset. The ‘enumerate’ function is used to add an index to each score so that we can sort them later.

```
similarity_scores.sort(key=lambda x: x[1], reverse=True)
```

This line sorts the similarity scores in descending order (i.e., from most similar to least similar) and returns a list of tuples where each tuple contains the index of a movie and its similarity score.

We then obtained the top 5 recommendations by taking the titles of the movies with the highest similarity scores.

```
# Get the top 5 movie recommendations for movie with id n

top_5_indices = [x[0] for x in similarity_scores[1:6]]

top_5_recommendations = movies.iloc[top_5_indices]['title'].tolist()

top_5_genres = movies.iloc[top_5_indices]['genres'].tolist()

for i in range(len(top_5_recommendations)):

    print(f"{i+1}. {top_5_recommendations[i]} ({top_5_genres[i]})")
```

Output:

```
1. Sabrina (1995) (Comedy|Romance)

2. Clueless (1995) (Comedy|Romance)

3. Two if by Sea (1996) (Comedy|Romance)

4. French Twist (Gazon maudit) (1995) (Comedy|Romance)

5. If Lucy Fell (1996) (Comedy|Romance)
```

From our above iteration we can see the genres recommended are in the Comedy|Romance genres.

2. Collaborative Filtering

The Collaborative Filtering Recommender is entirely based on the past behaviour and not on the context.It is based on the similarity in preferences, tastes and choices of two users.

It analyses how similar the tastes of one user is to another and makes recommendations on the basis of that.

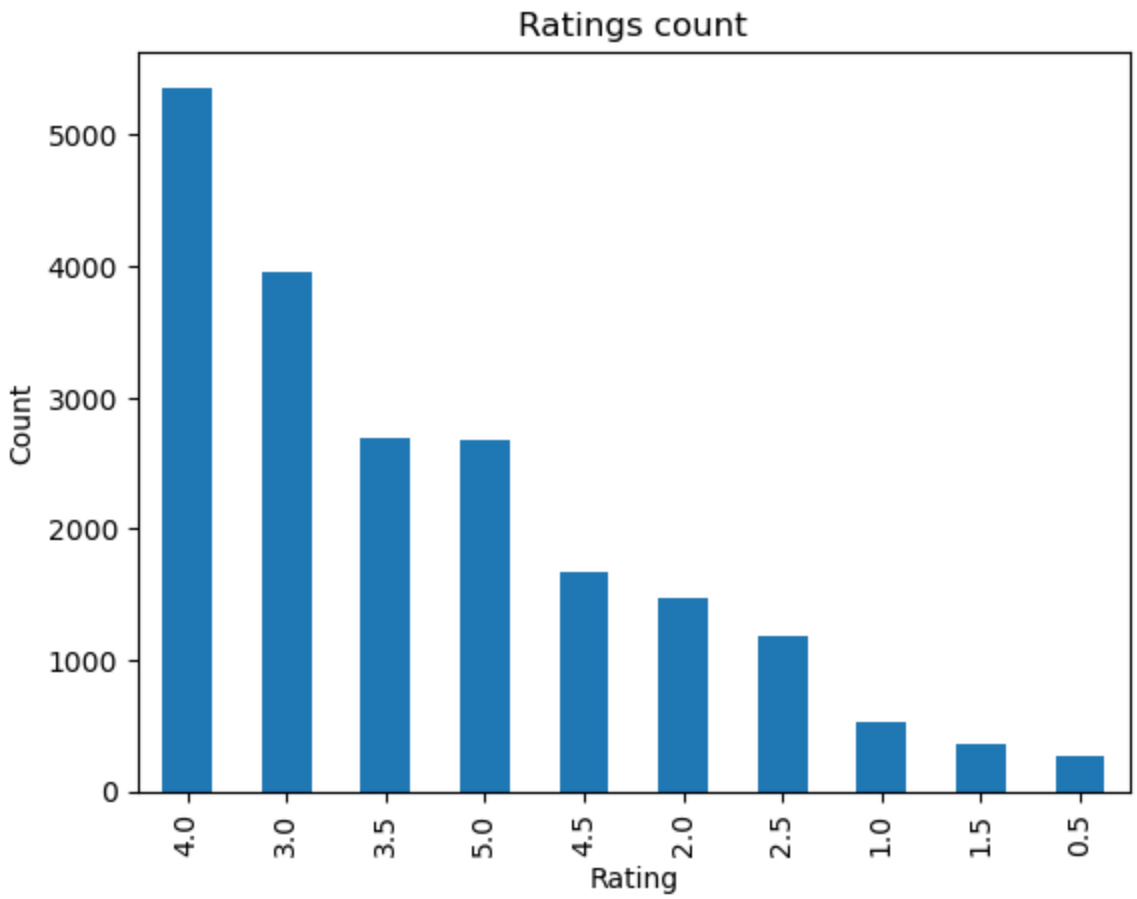
For instance, if user A likes movies 1, 2, 3 and user B likes movies 2,3,4, then they have similar interests and A should like movie 4 and B should like movie 1.

These recommendations can be acquired with two broad categories:

Memory-Based Collaborative Filtering (Neighbourhood based).

Model-Based Collaborative filtering.

Assume there are some users who have bought certain items, we can use a matrix with size num_users∗num_items to denote the past behaviour of users.
Each cell in the matrix represents the associated opinion that a user holds, such a matrix is called a **Utility Matrix**.



From our visual above we are able to deduce the different movie ratings against the movie count.

We are then going to split our dataset into train and test sets using surprise for implementing our model.

```
# Split into train and test set
from surprise import Reader, Dataset, SVD
from surprise.model_selection import train_test_split
df = pd.DataFrame(small_data, columns=['userId', 'movieId', 'rating'])
reader = Reader()
data = Dataset.load_from_df(df, reader)
trainset, testset = train_test_split(data, test_size=0.20)
```

Note: The 'trainset' still being in a 'suprise' specific data type means it has been optimized for
computational efficiency and the test set is a standard python list.

1. Memory-Based Methods(Neighborhood-Based)

Surprise gives you a chance to try out multiple different types of collaborative filtering engines.
One of our first decisions is item-item similarity versus user-user similarity.
In a case where we have fewer items than users, it will be more efficient to calculate item-item
similarity rather than user-user.

```
Number of users: 610
 Number of items: 4615
```

From our dataset above we can see our number of users is less than that of the items. We know for
the sake of computation time, its best to calculate the similarity between whichever number is
fewer - which in our case is users

a.)

```
#To train our model
sim_cos ={"name": "cosine", "user_based":True}
basic = knns.KNNBasic(sim_options=sim_cos)
basic.fit(trainset)
```

We shall test our model on how well it performed below by obtaining the RSME (root mean square
error)

```
predictions = basic.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 1.0523
 1.052302013721698
```

As you can see, the model had an RMSE of about 1.0711, meaning that it was off by roughly 1
point for each guess it made for ratings.
An RSME value of zero would indicate a perfect fit to our data. Let's try with a different similarity
metric (Pearson correlation) and evaluate our RMSE and see if our accuracy will improve.

b.)

```
# Using a different similarity metrics (Pearson correlation)
sim_pearson = {"name":"pearson","user_based":True}
basic_pearson = knns.KNNBasic(sim_options=sim_pearson)
basic_pearson.fit(trainset)
predictions = basic_pearson.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 1.0812
1.0811546553513849
```

Pearson correlation seems to have performed worse than cosine similarity with an RSME of 1.1044
compared to our previous one that was 1.0711 respectively. We can go ahead and use Cosine
similarity as our similarity metric of choice.

c.)

KNN with Means(basic KNN model) takes into account the mean rating of each user or item depending on whether you are performing user-user or item-item similarities.

```
# KNN with Means
sim_pearson = {'name':'pearson','user_based':True}
knn_means = knns.KNNWithMeans(sim_options = sim_pearson)
knn_means.fit(trainset)
predictions = knn_means.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 1.0072
1.0071579551654226
```

d.)

KNN Baseline model is more advanced as it adds in a bias term that is calculated by way of minimizing a cost function

```
sim_pearson = {'name':'pearson','user-based':True}
knn_baseline = knns.KNNBaseline(sim_options=sim_pearson)
knn_baseline.fit(trainset)
predictions = knn_baseline.test(testset)
```

```
print(accuracy.rmse(predictions))
```

```
RMSE: 0.9730
0.9730098574157473
```

Even better!

2. Model-Based Methods(Matrix factorization)

When SVD is calculated for recommendation systems, it only takes into account the rated values,
ignoring whatever items have not been rated by users.

```
from surprise.model_selection import GridSearchCV
param_grid = {'n_factors':[20, 100],'n_epochs': [5, 10], 'lr_all': [0.002, 0.005],'reg_a
gs_model = GridSearchCV(SVD,param_grid=param_grid,n_jobs = -1,joblib_verbose=5)
gs_model.fit(data)
```

The optimal parameters used are :

```
{'n_factors': 100, 'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.4}
```

```
# Using the optimal parameters from above
svd = SVD(n_factors=100, n_epochs=10, lr_all=0.005, reg_all=0.4)
svd.fit(trainset)
predictions = svd.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 0.9122
0.9121824084198218
```

In order to get predicted ratings for a given user and item, all that's needed are the userId and movieId for which you want to make a prediction on.

```
# Here we are making a prediction of user 42 and item 20 using the SVD we just fit
user_42_prediction = svd.predict("42","20")
user_42_prediction
```

```
Prediction(uid='42', iid='20', r_ui=None, est=3.5078100787206346,
details={'was_impossible':                                False})
```

Now using our predicted ratings for a given user, we are going to create a list of top 10 movies that we could recommend to that particular user.

```
# Load the ratings dataset using the Surprise library
reader = Reader(line_format='user item rating timestamp', sep=',',
skip_lines=1)
data = Dataset.load_from_file('Data_ratings.csv', reader=reader)
data
```

Our dataset will be in Surprise format.
Having trained our model from above using the 'svd.fit(trainset)' which is also in the format of an Surprise, we are going to use svd in our case as shown below.

```
# Get the user ratings for user with id 'n'
user_id = 3
user_ratings = data.raw_ratings
```

Here we will have a chance to look at the movies that that particular user did not rate so as to be able to make an informed judjement on what's best to recommend to them.

```
# Get all the movies that user with id 'n' has not rated yet
user_unrated_movies = movies[~movies['movieId'].isin([rating[1] for rating
in user_ratings if rating[0] == user_id])]

# Predict the ratings for all the unrated movies and sort them in
descending order
user_unrated_movies['predicted_rating'] =
user_unrated_movies['movieId'].apply(lambda x: svd.predict(user_id,
x).est)
user_unrated_movies.sort_values('predicted_rating', ascending=False,
inplace=True)

# Get the top 5 movie recommendations for user with id n
top_5_recommendations = user_unrated_movies.head()
top_5_recommendations
```

Output:

	movieId	title	genres	predicted_rating
277	318	Shawshank Redemption, The (1994)	Crime|Drama	3.963193
224	260	Star Wars: Episode IV - A New Hope (1977)	Action|Adventure|Sci-Fi	3.797867
461	527	Schindler's List (1993)	Drama|War	3.787157
898	1196	Star Wars: Episode V - The Empire Strikes Back...	Action|Adventure|Sci-Fi	3.728293
46	50	Usual Suspects, The (1995)	Crime|Mystery|Thriller	3.688113

Conclusion:

By leveraging both content-based and collaborative filtering techniques, we were able to develop a model that provided personalised movie recommendations to users based on their ratings on other movies and genres of preference.

Using the collaborative filtering approach specifically user-based, we identified similar users based on their movie ratings and genres which enabled us to generate recommendations based on the preference of users with similar tastes.

We utilized the surprise library which provided a framework for loading and preprocessing the data, splitting it into training and testing sets and implementing the algorithm. Our model was trained using evaluation metrics: RMSE to access its perfomance by measuring the accuracy, we ensured that our recommendations were reliable to our users.

By offering the top 5 movie recommendations to the users, we were able to enhance the movie viewage and experience and allow them to discover new films that intrigued them.

Recommendations:

Use of a hybrid recommendation systems that combines content-based and collaborative filtering, hence more accurate recommendations.
Provide a diverse selection of highly popular films that users may enjoy based on the ratings of other movies.