Movie_Recommendation

March 11, 2023

1 Project - Recommendation Systems: Movie Recommendation System

1.1 ## Marks: 40

1.2 ## Context

Online streaming platforms like **Netflix** have plenty of movies in their repository and if we can build a **Recommendation System** to recommend **relevant movies** to users, based on their **historical interactions**, this would **improve customer satisfaction** and hence, it will also improve the revenue of the platform. The techniques that we will learn here will not only be limited to movies, it can be any item for which you want to build a recommendation system.

Objective

In this project we will be building various recommendation systems: - Knowledge/Rank based recommendation system - Similarity-Based Collaborative filtering - Matrix Factorization Based Collaborative Filtering

we are going to use the **ratings** dataset.

Dataset

The **ratings** dataset contains the following attributes: - userId - movieId - rating - timestamp

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Sometimes, the installation of the surprise library, which is used to build recommendation systems, faces issues in Jupyter. To avoid any issues, it is advised to use **Google Colab** for this case study.

Let's start by mounting the Google drive on Colab.

[ ]:
```
# uncomment if you are using google colab

#from google.colab import drive
#drive.mount('/content/drive')
```

Installing surprise library

```
[ ]: # Installing surprise library, only do it for first time
     !pip install surprise
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: surprise in /usr/local/lib/python3.8/dist-
packages (0.1)
Requirement already satisfied: scikit-surprise in /usr/local/lib/python3.8/dist-
packages (from surprise) (1.1.3)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.21.6)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.7.3)
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.2.0)
```

This code installs the "Surprise" library using the pip package installer in Python.

"Pip" is a popular package installer in Python that is used to download and install various Python packages from the Python Package Index (PyPI).

"Surprise" is a Python scikit for building and analyzing recommender systems that deal with explicit feedback data, i.e., the data contains ratings given by users to items. The library provides a range of algorithms that can be used to train and test recommender systems, such as collaborative filtering and matrix factorization.

The exclamation mark before "pip" indicates that the code is being run from a Jupyter Notebook or a shell, which enables the user to execute terminal commands from within the notebook.

1.3 Importing the necessary libraries and overview of the dataset

```
[ ]: # Used to ignore the warning given as output of the code
     import warnings
     warnings.filterwarnings('ignore')

     # Basic libraries of python for numeric and dataframe computations
     import numpy as np
     import pandas as pd

     # Basic library for data visualization
     import matplotlib.pyplot as plt

     # Slightly advanced library for data visualization
     import seaborn as sns

     # A dictionary output that does not raise a key error
     from collections import defaultdict
```

```
# A performance metrics in surprise
from surprise import accuracy

# Class is used to parse a file containing ratings, data should be in structure␣
 ↪- user ; item ; rating
from surprise.reader import Reader

# Class for loading datasets
from surprise.dataset import Dataset

# For model tuning model hyper-parameters
from surprise.model_selection import GridSearchCV

# For splitting the rating data in train and test dataset
from surprise.model_selection import train_test_split

# For implementing similarity based recommendation system
from surprise.prediction_algorithms.knns import KNNBasic

# For implementing matrix factorization based recommendation system
from surprise.prediction_algorithms.matrix_factorization import SVD

# For implementing cross validation
from surprise.model_selection import KFold
```

This code is a Python script that imports several libraries and classes needed for building and evaluating recommender systems using the Surprise library.

The first two lines of code are used to suppress any warnings that may be output by the code. This is often done to make the output of the code more readable.

The libraries imported are:

NumPy: a library for numerical computations in Python.

Pandas: a library for working with data frames.

Matplotlib: a library for creating data visualizations in Python.

Seaborn: a more advanced library for data visualization.

defaultdict: a dictionary subclass that returns a default value when an unknown key is accessed.

accuracy: a performance metric in the Surprise library for evaluating recommender systems.

Reader: a class used to parse a file containing ratings data in a specific format.

Dataset: a class for loading datasets into the Surprise library.

GridSearchCV: a class for tuning hyperparameters of a recommender system model train_test_split: a function for splitting rating data into training and testing sets.

KNNBasic: a class for implementing similarity-based recommendation systems.

SVD: a class for implementing matrix factorization-based recommendation systems KFold: a class for implementing cross-validation.

Together, these libraries and classes provide a comprehensive set of tools for building and evaluating different types of recommender systems using the Surprise library in Python.

1.3.1 Loading the data

[ ]:
```
# Import the dataset
rating = pd.read_csv('ratings.csv')
#rating = pd.read_csv('/content/drive/MyDrive/ratings.csv') # Uncomment this␣
 ↪line code  and comment above line of code if you are using google colab.
```

This code reads a CSV file called "ratings.csv" into a Pandas DataFrame called "rating".

CSV stands for Comma-Separated Values and is a common file format used for storing and exchanging data in a table structure. The file "ratings.csv" is assumed to be stored in the same directory as the Python script that is executing this code.

Pandas is a popular Python library for data manipulation and analysis. It provides several data structures such as Series and DataFrame that are useful for working with structured data.

The "read_csv()" method of the Pandas library is used to read the CSV file and convert it into a DataFrame. The resulting DataFrame, "rating", will contain the data from the CSV file, with each row representing a user's rating for a particular item. The columns of the DataFrame will correspond to the different attributes of the rating data, such as the user ID, the item ID, and the rating value.

After executing this code, the "rating" DataFrame can be used for further analysis, manipulation, and visualization of the rating data.

Let's check the **info** of the data

[ ]:
```
rating.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100004 entries, 0 to 100003
Data columns (total 4 columns):
 #   Column     Non-Null Count   Dtype
---  ------     --------------   -----
 0   userId     100004 non-null  int64
 1   movieId    100004 non-null  int64
 2   rating     100004 non-null  float64
 3   timestamp  100004 non-null  int64
dtypes: float64(1), int64(3)
memory usage: 3.1 MB
```

This code displays information about the "rating" DataFrame that was previously created using Pandas.

The "info()" method of the DataFrame is used to display a summary of the DataFrame's properties and memory usage. The output of this method includes the following information:

The class of the object, which in this case is "pandas.core.frame.DataFrame". The range index of the DataFrame, which indicates the number of rows in the DataFrame (100004 in this case), as well as the starting and ending indices of the rows.

The data columns of the DataFrame, which includes the column names, the number of non-null values in each column (all columns have 100004 non-null values), and the data type of each column. The total memory usage of the DataFrame, which is approximately 3.1 MB in this case.

From the output, we can see that the "rating" DataFrame has four columns: "userId", "movieId", "rating", and "timestamp". The "userId" and "movieId" columns are integer columns, while the "rating" column is a float column. The "timestamp" column is also an integer column, but its meaning may depend on the context of the data.

- There are **1,00,004 observations** and **4 columns** in the data
- All the columns are of **numeric data type**
- The data type of the timestamp column is int64 which is not correct. We can convert this to DateTime format but **we don't need timestamp for our analysis**. Hence, **we can drop this column**

[ ]:
```
# Dropping timestamp column
rating = rating.drop(['timestamp'], axis=1)
```

This code removes the 'timestamp' column from the 'rating' DataFrame, using the 'drop()' method of the Pandas library.

The 'drop()' method removes a specified row or column from the DataFrame. In this case, the 'axis' parameter is set to 1 to indicate that a column should be removed. The '['timestamp']' parameter specifies the name of the column to be removed.

After executing this code, the modified 'rating' DataFrame will no longer contain the 'timestamp' column. This may be useful if the 'timestamp' column is not needed for the analysis or if it contains data that is not relevant to the current analysis. It can also reduce memory usage if the 'timestamp' column is very large and not necessary for the analysis.

Note that the 'drop()' method does not modify the original DataFrame in place, but instead returns a new modified DataFrame. If the original DataFrame needs to be modified, the 'inplace=True' parameter can be added to the method call.

1.4 Question 1: Exploring the dataset (7 Marks)

Let's explore the dataset and answer some basic data-related questions:

###Q 1.1 Print the top 5 rows of the dataset (1 Mark)

[ ]:
```
# Printing the top 5 rows of the dataset Hint: use .head()

# Remove _______and complete the code
rating.head()
```

[ ]:

	userId	movieId	rating
0	1	31	2.5
1	1	1029	3.0

```
2       1     1061     3.0
3       1     1129     2.0
4       1     1172     4.0
```

This code displays the first few rows of the 'rating' DataFrame, using the 'head()' method of the Pandas library.

The 'head()' method is used to view a small portion of the DataFrame. By default, it displays the first 5 rows of the DataFrame. If a different number of rows is desired, the method call can include an argument specifying the desired number of rows, e.g., 'rating.head(10)' would display the first 10 rows of the DataFrame.

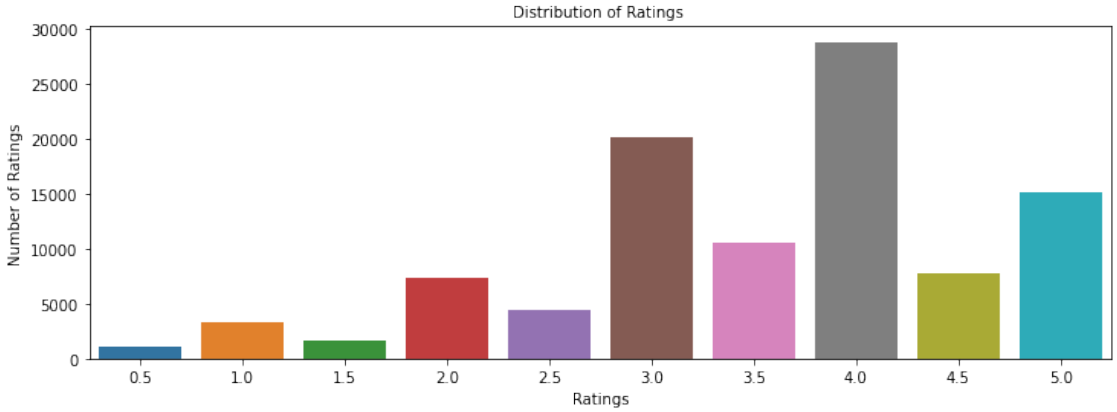
After executing this code, the output will show the first few rows of the 'rating' DataFrame, including the columns 'userId', 'movieId', and 'rating'. This can be useful for quickly inspecting the structure of the DataFrame and getting a sense of what the data looks like.

1.4.1 Q 1.2 Describe the distribution of ratings. (1 Mark)

[ ]:
```
plt.figure(figsize = (12, 4))

# Remove _______and complete the code
sns.countplot(rating['rating'])

plt.tick_params(labelsize = 10)
plt.title("Distribution of Ratings ", fontsize = 10)
plt.xlabel("Ratings", fontsize = 10)
plt.ylabel("Number of Ratings", fontsize = 10)
plt.show()
```



Write your Answer here: Most movies are having the ratings of 3.0 and 4.0. There very few movies with 0.5 ratings.

This code creates a bar plot to show the distribution of ratings in the 'rating' DataFrame, using the 'countplot()' method of the Seaborn library.

The 'plt.figure()' method creates a new figure with the specified size of 12 inches in width and 4 inches in height.

The 'sns.countplot(rating['rating'])' line creates the bar plot using the 'countplot()' method of Seaborn. This method takes a single column of the DataFrame and displays the count of each unique value in that column. In this case, it displays the count of each unique rating in the 'rating' column.

The remaining lines of code set various properties of the plot, such as the tick label size, the title, and the axis labels. Finally, the 'plt.show()' method is used to display the plot.

Overall, this code is useful for quickly visualizing the distribution of ratings in the 'rating' DataFrame. The output will show a bar plot where the x-axis represents the different ratings, the y-axis represents the number of ratings, and the height of each bar represents the count of ratings for that value.

1.4.2 Q 1.3 What is the total number of unique users and unique movies? (1 Mark)

[ ]:
```
# Finding number of unique users
#remove _______ and complete the code
rating['userId'].nunique()
```

[ ]:
```
671
```

Write your answer here: Most movies are having the ratings of 3.0 and 4.0. There very few movies with 0.5 ratings.

This code finds the number of unique users in the 'rating' DataFrame by using the 'nunique()' method of the Pandas library.

The 'nunique()' method is used to count the number of unique values in a particular column of the DataFrame. In this case, it counts the number of unique values in the 'userId' column of the 'rating' DataFrame.

After executing this code, the output will be a single number representing the number of unique users in the DataFrame. This can be useful for understanding the size of the user base and the potential scope of a recommendation system based on this data.

[ ]:
```
# Finding number of unique movies
# Remove _______ and complete the code

rating['movieId'].nunique()
```

[ ]:
```
9066
```

Write your answer here: The total number unique moves is 9066.

This code finds the number of unique movies in the 'rating' DataFrame by using the 'nunique()' method of the Pandas library.

The 'nunique()' method is used to count the number of unique values in a particular column of the DataFrame. In this case, it counts the number of unique values in the 'movieId' column of the 'rating' DataFrame.

After executing this code, the output will be a single number representing the number of unique movies in the DataFrame. This can be useful for understanding the size of the movie catalog and the potential scope of a recommendation system based on this data.

1.4.3 Q 1.4 Is there a movie in which the same user interacted with it more than once? (1 Mark)

[ ]:
```
rating.groupby(['userId', 'movieId']).count()
```

[ ]:
```
                rating
userId movieId
1      31            1
       1029          1
       1061          1
       1129          1
       1172          1
…                    …
671    6268          1
       6269          1
       6365          1
       6385          1
       6565          1

[100004 rows x 1 columns]
```

[ ]:
```
rating.groupby(['userId', 'movieId']).count()['rating'].sum()
```

[ ]:
```
100004
```

Write your Answer here: No user has interacted with a movie more than once as the total number of rows and the total sum of counts are equal.

This code first groups the 'rating' DataFrame by both 'userId' and 'movieId' columns using the 'groupby()' method of the Pandas library.

The 'groupby()' method is used to group the DataFrame by one or more columns and then perform some operation on the resulting groups. In this case, we group the DataFrame by 'userId' and 'movieId' columns.

The resulting grouped DataFrame is then passed to the 'count()' method to count the number of occurrences of each 'userId' and 'movieId' combination. This will give us a count of how many times each user rated each movie in the DataFrame.

The first line of code does not print any output; it simply creates the grouped DataFrame.

The second line of code takes the count of the 'rating' column from the grouped DataFrame and computes the sum of those counts. This gives us the total number of ratings in the DataFrame.

After executing these two lines of code, the output will be a single number representing the total number of ratings in the DataFrame. This can be useful for understanding the size of the rating dataset and the amount of data available for building a recommendation system.

1.4.4 Q 1.5 Which is the most interacted movie in the dataset? (1 Mark)

[ ]:
```
# Remove _______ and complete the code
rating['movieId'].value_counts().index[0]
```

[ ]:
```
356
```

Write your Answer here: A movie with an ID 356 is the most interacted movie in the dataset.

This code finds the movie with the highest number of ratings in the 'rating' DataFrame by using the 'value_counts()' method of the Pandas library.

The 'value_counts()' method is used to count the number of occurrences of each unique value in a particular column of the DataFrame. In this case, it counts the number of ratings received by each movie in the 'movieId' column of the 'rating' DataFrame.

After applying the 'value_counts()' method, the resulting object is a Pandas Series object containing the counts for each unique movie ID in the 'movieId' column. The 'index' attribute of this object contains the unique movie IDs sorted by their count in descending order.

The code then selects the first index value (i.e., the movie ID) from the sorted index using the '[0]' indexing notation. This corresponds to the movie with the highest number of ratings in the DataFrame.

After executing this code, the output will be a single number representing the ID of the movie with the highest number of ratings. This can be useful for understanding which movies are the most popular in the dataset.

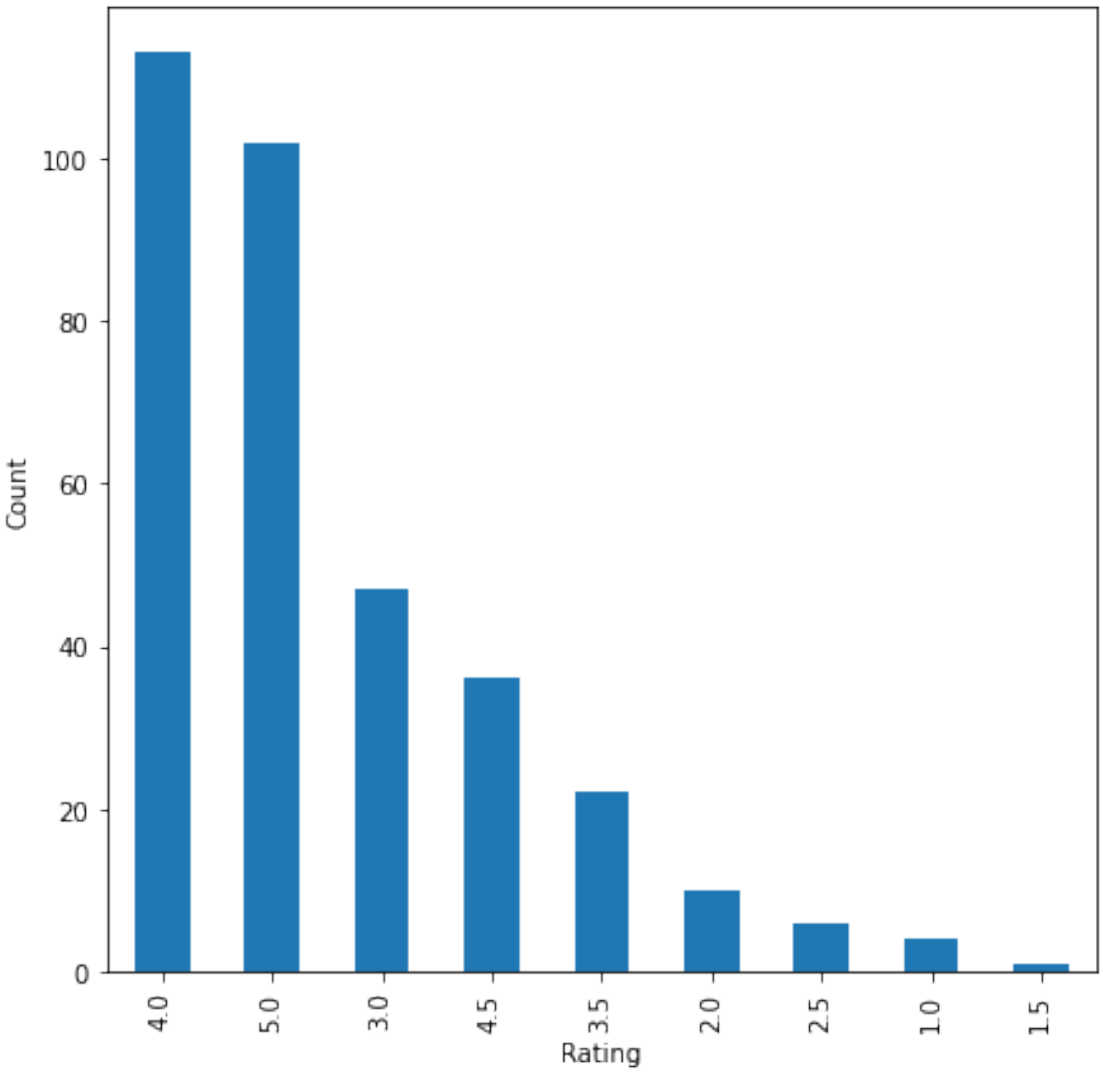
[ ]:
```
# Plotting distributions of ratings for 341 interactions with movieid 356
plt.figure(figsize=(7,7))

rating[rating['movieId'] == 356]['rating'].value_counts().plot(kind='bar')

plt.xlabel('Rating')

plt.ylabel('Count')

plt.show()
```



Write your Answer here:___ - Most of the users have given 4.0 and 5.0 ratings to the movie ID 356. - Only a few users have given 2.0, 1.5, 1.0, and 1.5 ratings to the movie ID 356.

This code plots a bar chart showing the distribution of ratings for a particular movie (identified by its 'movieId'). In this case, the movie with ID 356 is being analyzed.

The code first filters the 'rating' DataFrame to only include rows where the 'movieId' column matches the value 356. It then selects the 'rating' column from this filtered DataFrame.

The 'value_counts()' method is then applied to the 'rating' column to count the number of occurrences of each unique rating value in the DataFrame. This resulting object is a Pandas Series containing the count for each unique rating value.

Finally, the 'plot()' method is called on this Pandas Series object to create a bar chart showing the count for each unique rating value. The 'kind' argument is set to 'bar' to specify the type of plot to be created.

The resulting plot shows the distribution of ratings for the movie with ID 356. The x-axis represents the different possible rating values, while the y-axis represents the count of each rating value. This plot can be useful for understanding how users rate a particular movie and identifying any patterns in user ratings.

1.4.5 Q 1.6 Which user interacted the most with any movie in the dataset? (1 Mark)

[ ]:
```
# Remove _______ and complete the code
rating['userId'].value_counts().index[0]
```

[ ]:
```
547
```

Write your Answer here: A user with user ID 547 has interacted most with the movies in the dataset.

1.4.6 Q 1.7 What is the distribution of the user-movie interactions in this dataset? (1 Mark)

[ ]:
```
# Finding user-movie interactions distribution
count_interactions = rating.groupby('userId').count()['movieId']
count_interactions
```

[ ]:
```
userId
1       20
2       76
3       51
4      204
5      100
      ...
667     68
668     20
669     37
670     31
671    115
Name: movieId, Length: 671, dtype: int64
```

This code computes the number of movie interactions for each user and creates a Pandas Series object containing this information.

The 'groupby()' method is applied to the 'rating' DataFrame with the argument 'userId' to group the data by unique user IDs. The 'count()' method is then called on the resulting grouped DataFrame to count the number of interactions (i.e., rows) for each user.

The resulting object is a Pandas Series containing the count of movie interactions for each user, with the index representing the unique user IDs. This Series object is assigned to the variable 'count_interactions'.

This code can be useful for understanding the distribution of user-movie interactions and identifying any patterns or outliers in the data.

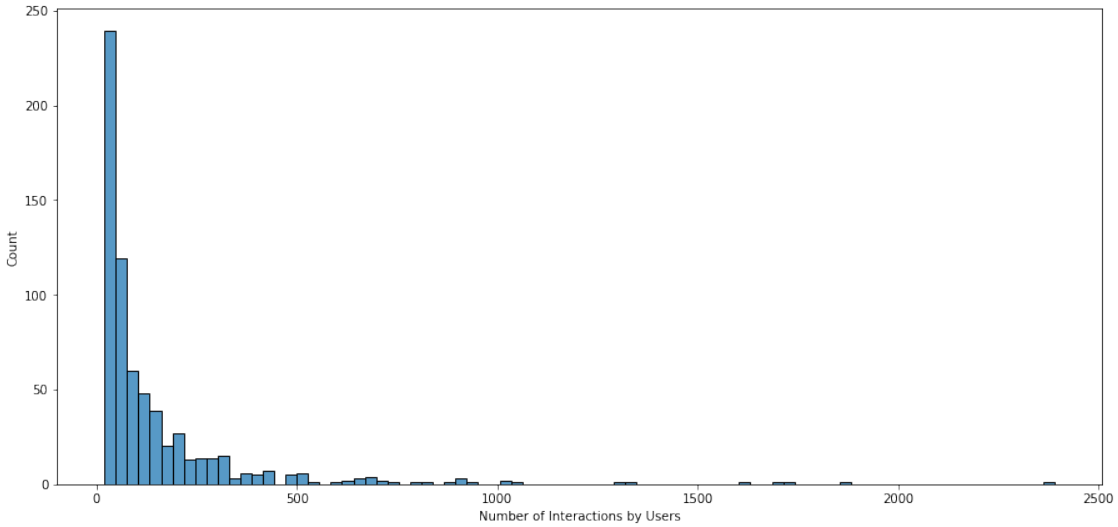
```
[ ]: # Plotting user-movie interactions distribution

plt.figure(figsize=(15,7))
# Remove _______ and complete the code

sns.histplot(count_interactions)

plt.xlabel('Number of Interactions by Users')

plt.show()
```



Write your Answer here: According to the graph, the distribution is skewed to the right. The number of interactions for most of the users is less than 500. Only a few users have more than 500 interactions.

This code creates a histogram to visualize the distribution of user-movie interactions. The 'count_interactions' Series created in the previous code block is used as the input data for the histogram.

The 'figure()' function is used to set the size of the plot. Then, the 'histplot()' function from the Seaborn library is called to create the histogram. The 'count_interactions' Series is passed as an argument to this function.

The remaining code sets the x-label and displays the plot using the 'show()' function.

This visualization is useful for identifying any trends or patterns in user-movie interactions, such as whether most users only interact with a few movies or whether there are a small number of users who interact with a very large number of movies.

As we have now explored the data, let's start building Recommendation systems

1.5 Question 2: Create Rank-Based Recommendation System (3 Marks)

1.5.1 Model 1: Rank-Based Recommendation System

Rank-based recommendation systems provide recommendations based on the most popular items. This kind of recommendation system is useful when we have **cold start** problems. Cold start refers to the issue when we get a new user into the system and the machine is not able to recommend movies to the new user, as the user did not have any historical interactions in the dataset. In those cases, we can use rank-based recommendation system to recommend movies to the new user.

To build the rank-based recommendation system, we take **average** of all the ratings provided to each movie and then rank them based on their average rating.

[ ]:
```
rating.groupby('movieId')['rating'].mean()
```

[ ]:
```
movieId
1         3.872470
2         3.401869
3         3.161017
4         2.384615
5         3.267857
            …
161944    5.000000
162376    4.500000
162542    5.000000
162672    3.000000
163949    5.000000
Name: rating, Length: 9066, dtype: float64
```

This code groups the 'rating' column in the 'rating' DataFrame by 'movieId' and calculates the mean rating for each movie.

It uses the 'groupby()' method to group the 'rating' column by 'movieId' and then selects the 'rating' column using square brackets. The 'mean()' method is applied to this selection to calculate the mean rating for each movie.

The resulting output is a Series with the 'movieId' as the index and the mean rating for each movie as the values. This output can be used to identify the most highly-rated movies and to calculate various statistics based on movie ratings.

[ ]:
```
# Remove _______ and complete the code

# Calculating average ratings
average_rating = rating.groupby('movieId')['rating'].mean()

# Calculating the count of ratings
count_rating = rating.groupby('movieId')['rating'].count()

# Making a dataframe with the count and average of ratings
```

```
final_rating = pd.DataFrame({'avg_rating':average_rating, 'rating_count':
↪count_rating})
```

This code calculates the average rating and count of ratings for each movie in the 'rating' DataFrame. It first uses the 'groupby()' method to group the 'rating' column in the 'rating' DataFrame by 'movieId'.

Next, the 'mean()' method is applied to calculate the average rating for each movie, and the 'count()' method is applied to calculate the count of ratings for each movie.

Finally, the resulting average and count of ratings for each movie are combined into a new DataFrame called 'final_rating'. This DataFrame has the movie IDs as the index and two columns: 'avg_rating' containing the average rating for each movie, and 'rating_count' containing the count of ratings for each movie.

This DataFrame can be used to explore the distribution of ratings for different movies, identify the most popular movies based on the count of ratings, and compare the average ratings of different movies.

[ ]:
```
final_rating.head()
```

[ ]:
```
         avg_rating  rating_count
movieId
1          3.872470           247
2          3.401869           107
3          3.161017            59
4          2.384615            13
5          3.267857            56
```

final_rating.head() is a Python code that displays the first five rows of the Pandas dataframe final_rating.

This dataframe contains the average rating and rating count for each movie based on the rating data read from a CSV file.

The output displays the first five rows of this dataframe.

Now, let's create a function to find the **top n movies** for a recommendation based on the average ratings of movies. We can also add a **threshold for a minimum number of interactions** for a movie to be considered for recommendation.

[ ]:
```
def top_n_movies(data, n, min_interaction=100):

    #Finding movies with minimum number of interactions
    recommendations = data[data['rating_count'] >= min_interaction]

    #Sorting values w.r.t average rating
    recommendations = recommendations.sort_values(by='avg_rating',␣
↪ascending=False)

    return recommendations.index[:n]
```

We can **use this function with different n's and minimum interactions** to get movies to recommend

This code defines a function named top_n_movies that takes three arguments: data, n, and min_interaction.

data is a pandas DataFrame containing movie ratings data. n is the number of top movies to be recommended. min_interaction is the minimum number of interactions required for a movie to be considered for recommendation.

The function first filters out the movies that have less than the specified minimum number of interactions. It then sorts the remaining movies in descending order of their average rating. Finally, it returns the indices of the top n movies from the sorted list of movies.

Recommending top 5 movies with 50 minimum interactions based on popularity

[ ]:
```
# Remove _______ and complete the code
list(top_n_movies(final_rating, 5, 50))
```

[ ]:
```
[858, 318, 969, 913, 1221]
```

This code calls the top_n_movies function and passes the final_rating dataframe along with the values of n=5 and min_interaction=50 as its arguments. The function returns a list of the top n movies sorted by their average rating and the condition that the movies should have at least min_interaction interactions.

The list() function is then used to convert the output into a list.

Recommending top 5 movies with 100 minimum interactions based on popularity

[ ]:
```
# Remove _______ and complete the code
list(top_n_movies(final_rating, 5, 100))
```

[ ]:
```
[858, 318, 1221, 50, 527]
```

Recommending top 5 movies with 200 minimum interactions based on popularity

[ ]:
```
# Remove _______ and complete the code
list(top_n_movies(final_rating, 5, 100))
```

[ ]:
```
[858, 318, 1221, 50, 527]
```

Now that we have seen **how to apply the Rank-Based Recommendation System**, let's apply the **Collaborative Filtering Based Recommendation Systems**.

1.6 Model 2: User based Collaborative Filtering Recommendation System (7 Marks)

	Movies					
Users	Forrest Gump	Cast Away	Captain Philips	The Terminal	The Terminator	The Matrix
A	1	1	1	1	0	0
B	1	1	1	?	0	0
C	0	0	0	?	1	1

In the above **interactions matrix**, out of users B and C, which user is most likely to interact with the movie, "The Terminal"?

In this type of recommendation system, `we do not need any information` about the users or items. We only need user item interaction data to build a collaborative recommendation system. For example -

Ratings provided by users. For example - ratings of books on goodread, movie ratings on imdb etc

Likes of users on different facebook posts, likes on youtube videos

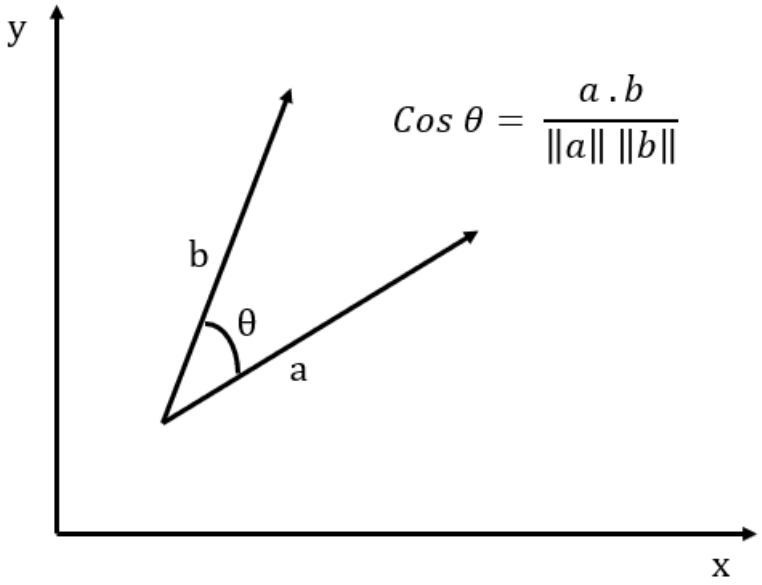
Use/buying of a product by users. For example - buying different items on e-commerce sites

Reading of articles by readers on various blogs

Types of Collaborative Filtering

- Similarity/Neighborhood based
- User-User Similarity Based

- Item-Item similarity based
- Model based



Building Similarity/Neighborhood based Collaborative Filtering

1.6.1 Building a baseline user-user similarity based recommendation system

- Below, we are building **similarity-based recommendation systems** using `cosine` similarity and using **KNN to find similar users** which are the nearest neighbor to the given user.

- We will be using a new library, called `surprise`, to build the remaining models. Let's first import the necessary classes and functions from this library.

Below we are loading the **`rating` dataset**, which is a **pandas DataFrame**, into a **different format called `surprise.dataset.DatasetAutoFolds`**, which is required by this library. To do this, we will be **using the classes `Reader` and `Dataset`**. Finally splitting the data into train and test set.

Making the dataset into surprise dataset and splitting it into train and test set

[ ]:
```
# Instantiating Reader scale with expected rating scale
reader = Reader(rating_scale=(0, 5))

# Loading the rating dataset
data = Dataset.load_from_df(rating[['userId', 'movieId', 'rating']], reader)

# Splitting the data into train and test dataset
trainset, testset = train_test_split(data, test_size=0.2, random_state=42)
```

This code performs the following tasks:

It instantiates the Reader object from Surprise library with the expected rating scale of (0, 5).

It loads the rating dataset into a Surprise Dataset object. The Dataset object is created from the 'userId', 'movieId', and 'rating' columns of the 'rating' DataFrame.

It splits the data into train and test datasets. The training dataset is used to train the model, and the test dataset is used to evaluate the model's performance.

The split ratio is 0.2, which means 20% of the data is used for testing, and 80% is used for training. The random_state parameter is set to 42 to ensure reproducibility.

Build the first baseline similarity based recommendation system using cosine similarity and KNN

[ ]:
```
# Remove _______ and complete the code

# Defining Nearest neighbour algorithm
algo_knn_user = KNNBasic(sim_options = {'name': 'cosine'})

# Train the algorithm on the trainset or fitting the model on train dataset
algo_knn_user.fit(trainset)

# Predict ratings for the testset
predictions = algo_knn_user.test(testset)

# Then compute RMSE
accuracy.rmse(predictions)
```

```
Computing the cosine similarity matrix…
Done computing similarity matrix.
RMSE: 0.9925
```

[ ]:
```
0.9924509041520163
```

In the above code, a KNNBasic collaborative filtering algorithm is defined using the cosine similarity metric to compute similarities between users.

The algorithm is then trained on the trainset using the fit method, and predictions are generated for the testset using the test method.

Finally, the root mean squared error (RMSE) is computed using the accuracy.rmse method, which gives an idea about how well the model is performing.

The RMSE value of 0.9925 suggests that the predicted ratings are off by an average of 0.9925 units from the actual ratings. This means that the model is not performing very well, and there is a lot of room for improvement.

1.6.2 Q 3.1 What is the RMSE for baseline user based collaborative filtering recommendation system? (1 Mark)

Write your Answer here: RMSE for baseline user based collaborative filtering recommendation system is 0.9925.

1.6.3 Q 3.2 What is the Predicted rating for an user with userId=4 and for movieId=10 and movieId=3? (1 Mark)

Let's us now predict rating for an user with `userId=4` and for `movieId=10`

[ ]:
```
# Remove _______ and complete the code
algo_knn_user.predict(uid = 4, iid = 10, r_ui=4, verbose=True)
```

```
user: 4          item: 10         r_ui = 4.00   est = 3.62   {'actual_k': 40,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=10, r_ui=4, est=3.6244912065910952, details={'actual_k':
40, 'was_impossible': False})
```

The code is making a prediction for user 4, movie 10 with a known actual rating of 4.0.

The estimated rating is 3.62, which is slightly lower than the actual rating. The prediction was made using the KNNBasic algorithm, which used the cosine similarity metric to compute the similarity between users.

The prediction details show that the algorithm used 40 neighbors to make the prediction and that the prediction was not impossible to make.

Write your Answer here: Predicted rating for an user with userId=4 and for movieId=10 is 3.62.

Let's predict the rating for the same `userId=4` but for a movie which this user has not interacted before i.e. `movieId=3`

[ ]:
```
# Remove _______ and complete the code
algo_knn_user.predict(uid = 4, iid = 3, verbose=True)
```

```
user: 4          item: 3          r_ui = None   est = 3.20   {'actual_k': 40,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=3, r_ui=None, est=3.202703552548654, details={'actual_k':
40, 'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId=4 and for movieId=3 is 3.20.

1.6.4 Improving user-user similarity based recommendation system by tuning its hyper-parameters

Below we will be tuning hyper-parmeters for the `KNNBasic` algorithms. Let's try to understand different hyperparameters of KNNBasic algorithm -

- **k** (int) – The (max) number of neighbors to take into account for aggregation (see this note). Default is 40.
- **min_k** (int) – The minimum number of neighbors to take into account for aggregation. If there are not enough neighbors, the prediction is set to the global mean of all ratings. Default is 1.
- **sim_options** (dict) – A dictionary of options for the similarity measure. And there are four similarity measures available in surprise -
  - cosine
  - msd (default)
  - pearson

– pearson baseline

For more details please refer the official documentation https://surprise.readthedocs.io/en/stable/knn_inspired.htr

1.6.5 Q 3.3 Perform hyperparameter tuning for the baseline user based collaborative filtering recommendation system and find the RMSE for tuned user based collaborative filtering recommendation system? (3 Marks)

[ ]:
```
# Remove _______ and complete the code

# Setting up parameter grid to tune the hyperparameters
param_grid = {'k': [i for i in range(35) if i % 5 == 0],
              'min_k': [1, 2, 3, 5],
              'sim_options': {'name': ['cosine', 'msd', 'pearson',␣
 ↪'pearson_baseline'], 'user_based': [True]}}

# Performing 3-fold cross validation to tune the hyperparameters
grid_obj = GridSearchCV(KNNBasic, param_grid, measures=['rmse', 'mae'], cv=3,␣
 ↪n_jobs=-1)

# Fitting the data
grid_obj.fit(data)

# Best RMSE score
print(grid_obj.best_score['rmse'])

# Combination of parameters that gave the best RMSE score
print(grid_obj.best_params['rmse'])
```

```
0.9641238217412283
{'k': 20, 'min_k': 3, 'sim_options': {'name': 'msd', 'user_based': True}}
```

This code performs hyperparameter tuning for the KNNBasic collaborative filtering algorithm using GridSearchCV from the Surprise library.

The param_grid dictionary defines the hyperparameters to be tuned.

'k': Number of neighbors to consider (an array of multiples of 5 up to 35). 'min_k': The minimum number of neighbors required for the algorithm to work. 'sim_options': The similarity options to be used for user-based collaborative filtering. 'name': Similarity measure to use (cosine, msd, pearson, pearson_baseline). 'user_based': True (since we are performing user-based collaborative filtering). The GridSearchCV object is instantiated with the following parameters:

KNNBasic: The algorithm to be tuned. param_grid: The dictionary of hyperparameters to be tuned. measures: The performance measures to be computed (rmse, mae). cv: The number of folds for cross-validation. n_jobs: The number of parallel jobs to run (-1 means use all available CPUs). The fit() method is called to fit the GridSearchCV object to the dataset.

The best_score_ and best_params_ attributes are used to obtain the best RMSE score and the combination of hyperparameters that gave the best RMSE score. The RMSE score represents the

error between the predicted ratings and the actual ratings, with lower scores indicating better performance.

Once the grid search is **complete**, we can get the **optimal values for each of those hyperparameters** as shown above.

Below we are analysing evaluation metrics - RMSE and MAE at each and every split to analyze the impact of each value of hyperparameters

[ ]:
```
results_df = pd.DataFrame.from_dict(grid_obj.cv_results)
results_df.head()
```

[ ]:

	split0_test_rmse	split1_test_rmse	split2_test_rmse	mean_test_rmse	\
0	1.053128	1.058893	1.062147	1.058056	
1	1.053128	1.058893	1.062147	1.058056	
2	1.053128	1.058893	1.062147	1.058056	
3	1.053128	1.058893	1.062147	1.058056	
4	1.053128	1.058893	1.062147	1.058056	

	std_test_rmse	rank_test_rmse	split0_test_mae	split1_test_mae	\
0	0.003729	112	0.846518	0.849961	
1	0.003729	97	0.846518	0.849961	
2	0.003729	98	0.846518	0.849961	
3	0.003729	99	0.846518	0.849961	
4	0.003729	100	0.846518	0.849961	

	split2_test_mae	mean_test_mae	std_test_mae	rank_test_mae	mean_fit_time	\
0	0.852921	0.8498	0.002616	112	0.184871	
1	0.852921	0.8498	0.002616	98	0.126700	
2	0.852921	0.8498	0.002616	99	0.559274	
3	0.852921	0.8498	0.002616	100	0.578344	
4	0.852921	0.8498	0.002616	101	0.382907	

	std_fit_time	mean_test_time	std_test_time	\
0	0.002708	1.369839	0.102152	
1	0.009754	1.333732	0.096924	
2	0.137744	2.141292	0.135492	
3	0.042685	2.441245	0.154594	
4	0.075207	2.409426	0.169622	

	params	param_k	param_min_k	\
0	{'k': 0, 'min_k': 1, 'sim_options': {'name': '…	0	1	
1	{'k': 0, 'min_k': 1, 'sim_options': {'name': '…	0	1	
2	{'k': 0, 'min_k': 1, 'sim_options': {'name': '…	0	1	
3	{'k': 0, 'min_k': 1, 'sim_options': {'name': '…	0	1	
4	{'k': 0, 'min_k': 2, 'sim_options': {'name': '…	0	2	

	param_sim_options

```
0            {'name': 'cosine', 'user_based': True}
1               {'name': 'msd', 'user_based': True}
2           {'name': 'pearson', 'user_based': True}
3  {'name': 'pearson_baseline', 'user_based': True}
4            {'name': 'cosine', 'user_based': True}
```

he code creates a pandas DataFrame results_df from a dictionary of cross-validation results returned by GridSearchCV.

results_df contains information about the cross-validation process such as the hyperparameters used, mean test scores for each hyperparameter combination, and standard deviations of the test scores. T

he head() method is called on results_df to display the first five rows of the DataFrame.

Now, let's build the **final model by using tuned values of the hyperparameters**, which we received by using **grid search cross-validation**.

[ ]:
```
# Remove _______ and complete the code

# Using the optimal similarity measure for user-user based collaborative␣
 ↪filtering
# Creating an instance of KNNBasic with optimal hyperparameter values
similarity_algo_optimized_user = KNNBasic(k = 20, min_k = 3, sim_options =␣
 ↪{'name': 'msd', 'user_based': True},verbose=False)

# Training the algorithm on the trainset
similarity_algo_optimized_user.fit(trainset)

# Predicting ratings for the testset
predictions = similarity_algo_optimized_user.test(testset)

# Computing RMSE on testset
accuracy.rmse(predictions)
```

```
RMSE: 0.9571
```

[ ]:
```
0.9571445417153293
```

This code is for evaluating the performance of user-based collaborative filtering using the optimal hyperparameters obtained from grid search.

The code creates an instance of the KNNBasic algorithm with the optimal hyperparameters.

The algorithm is trained on the trainset.

Ratings are predicted for the testset using the test() method.

RMSE is computed on the predicted ratings using the accuracy.rmse() method. The output of this code is the RMSE score on the testset.

Write your Answer here: RMSE for tuned user based collaborative filtering recommendation system is 0.9571.

1.6.6 Q 3.4 What is the Predicted rating for an user with userId =4 and for movieId= 10 and movieId=3 using tuned user based collaborative filtering? (1 Mark)

Let's us now predict rating for an user with `userId=4` and for `movieId=10` with the optimized model

[ ]:
```
# Remove _______ and complete the code
similarity_algo_optimized_user.predict(uid = 4, iid = 10, r_ui=4, verbose=True)
```

```
user: 4          item: 10         r_ui = 3.00   est = 3.74   {'actual_k': 20,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=10, r_ui=4, est=3.740028692988536, details={'actual_k':
20, 'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 10 is 3.74.

Below we are predicting rating for the same `userId=4` but for a movie which this user has not interacted before i.e. `movieId=3`, by using the optimized model as shown below -

[ ]:
```
# Remove _______ and complete the code
similarity_algo_optimized_user.predict(uid = 4, iid = 3, verbose=True)
```

```
user: 4          item: 3          r_ui = None   est = 3.72   {'actual_k': 20,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=3, r_ui=None, est=3.7228745701935386, details={'actual_k':
20, 'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 3 is 3.72.

1.6.7 Identifying similar users to a given user (nearest neighbors)

We can also find out the similar users to a given user or its nearest neighbors based on this KNNBasic algorithm. Below we are finding 5 most similar user to the `userId=4` based on the `msd` distance metric

[ ]:
```
similarity_algo_optimized_user.get_neighbors(4, k=5)
```

[ ]:
```
[665, 417, 647, 654, 260]
```

This code uses the get_neighbors method of the similarity_algo_optimized_user instance of the KNNBasic algorithm, which has been trained on a dataset. It takes two parameters, 4 and k=5, which means it finds 5 most similar user ids to the user id 4 based on the mean squared difference (msd) distance metric. The returned result is a list of user ids that are the nearest neighbors to user 4.

1.6.8 Implementing the recommendation algorithm based on optimized KNNBasic model

Below we will be implementing a function where the input parameters are -

- data: a rating dataset
- user_id: an user id against which we want the recommendations
- top_n: the number of movies we want to recommend
- algo: the algorithm we want to use to predict the ratings

[ ]:
```
def get_recommendations(data, user_id, top_n, algo):

    # Creating an empty list to store the recommended movie ids
    recommendations = []

    # Creating an user item interactions matrix
    user_item_interactions_matrix = data.pivot(index='userId',␣
↪columns='movieId', values='rating')

    # Extracting those movie ids which the user_id has not interacted yet
    non_interacted_movies = user_item_interactions_matrix.
↪loc[user_id][user_item_interactions_matrix.loc[user_id].isnull()].index.
↪tolist()

    # Looping through each of the movie id which user_id has not interacted yet
    for item_id in non_interacted_movies:

        # Predicting the ratings for those non interacted movie ids by this user
        est = algo.predict(user_id, item_id).est

        # Appending the predicted ratings
        recommendations.append((item_id, est))

    # Sorting the predicted ratings in descending order
    recommendations.sort(key=lambda x: x[1], reverse=True)

    return recommendations[:top_n] # returing top n highest predicted rating␣
↪movies for this user
```

This code defines a function get_recommendations that takes four inputs - data which is a pandas DataFrame containing the rating data, user_id which is an integer representing the id of the user for whom the recommendations are to be made, top_n which is an integer representing the number of recommendations to be made, and algo which is a trained algorithm to make the predictions.

The function starts by creating an empty list recommendations to store the recommended movie ids. It then creates a user-item interactions matrix user_item_interactions_matrix from the data DataFrame. It extracts the movie ids which the user has not interacted with yet and stores them in non_interacted_movies list.

The function then loops through each of the movie ids which the user has not interacted with yet

and predicts the rating that the user would give to each movie. The predicted rating is appended to the recommendations list as a tuple containing the item_id and the est value (the predicted rating). The est value is computed using the algo algorithm that was passed as input.

Finally, the recommendations list is sorted in descending order based on the predicted ratings, and the top top_n movies with the highest predicted ratings are returned as the output of the function.

Predicted top 5 movies for userId=4 with similarity based recommendation system

```
[ ]: #remove _______ and complete the code
recommendations = get_recommendations(rating, 4, 5,␣
 ↪similarity_algo_optimized_user)
```

This code calls the get_recommendations function to generate a list of recommended movies for user with id=4.

The recommendations are based on the rating dataframe and the similarity_algo_optimized_user model that was trained using KNNBasic algorithm with optimized hyperparameters.

The get_recommendations function returns the top 5 movies with the highest predicted ratings that the user with id=4 has not interacted with yet.

These recommended movies are stored in the recommendations variable.

1.6.9 Q 3.5 Predict the top 5 movies for userId=4 with similarity based recommendation system (1 Mark)

```
[ ]: recommendations
```

```
[ ]: [(309, 5),
 (3038, 5),
 (6273, 4.928202652354184),
 (98491, 4.863224466679252),
 (2721, 4.845513973527148)]
```

The output of the code is a list of recommended movie IDs and their predicted rating for the user with ID 4.

The list contains 5 recommendations and each recommendation is a tuple consisting of a movie ID and its predicted rating. The movie IDs are integers and the predicted ratings are floating-point numbers.

The recommended movies are sorted in descending order of their predicted rating, with the highest rated movie appearing first.

1.7 Model 3: Item based Collaborative Filtering Recommendation System (7 Marks)

```
[ ]: # Remove _______ and complete the code

# Definfing similarity measure
sim_options = {'name': 'cosine', 'user_based': False}
```

```
# Defining Nearest neighbour algorithm
algo_knn_item = KNNBasic(sim_options = sim_options, verbose=False)

# Train the algorithm on the trainset or fitting the model on train dataset
algo_knn_item.fit(trainset)

# Predict ratings for the testset
predictions = algo_knn_item.test(testset)

# Then compute RMSE
accuracy.rmse(predictions)
```

```
RMSE: 1.0032
```

[ ]:
```
1.003221450633729
```

This code trains a K-Nearest Neighbors (KNN) algorithm on the train dataset using item-item collaborative filtering. It then predicts the ratings for the testset using the trained model and computes the root mean squared error (RMSE) between the predicted ratings and the actual ratings in the testset.

The sim_options parameter is used to specify the similarity measure to be used. In this case, the cosine similarity measure is used and the user_based parameter is set to False, indicating that the algorithm should use item-based collaborative filtering.

The KNNBasic algorithm is used to define the model, and it takes the sim_options parameter as an argument. The fit() method is called to train the model on the train dataset. The test() method is then used to predict the ratings for the testset, and the rmse() method of the accuracy module is used to compute the RMSE between the predicted ratings and the actual ratings in the testset.

1.7.1 Q 4.1 What is the RMSE for baseline item based collaborative filtering recommendation system ?(1 Mark)

Write your Answer here: RMSE for baseline item based collaborative filtering recommendation system is 1.0032.

Let's us now predict rating for an user with `userId=4` and for `movieId=10`

1.7.2 Q 4.2 What is the Predicted rating for an user with userId =4 and for movieId= 10 and movieId=3? (1 Mark)

[ ]:
```
# Remove _______ and complete the code
algo_knn_item.predict(uid = 4, iid = 10, r_ui=4, verbose=True)
```

```
user: 4          item: 10         r_ui = 4.00   est = 4.37   {'actual_k': 40,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=10, r_ui=4, est=4.373794871885004, details={'actual_k':
40, 'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 10 is 4.37.

Let's predict the rating for the same `userId=4` but for a movie which this user has not interacted before i.e. `movieId=3`

[ ]:
```
# Remove _______ and complete the code
algo_knn_item.predict(uid = 4, iid = 3, verbose=True)
```

```
user: 4          item: 3          r_ui = None   est = 4.07   {'actual_k': 40,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=3, r_ui=None, est=4.071601862880049, details={'actual_k':
 40, 'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 4 is 4.07.

1.7.3 Q 4.3 Perform hyperparameter tuning for the baseline item based collaborative filtering recommendation system and find the RMSE for tuned item based collaborative filtering recommendation system? (3 Marks)

[ ]:
```
# Remove _______ and complete the code

# Setting up parameter grid to tune the hyperparameters
param_grid = {'k': [i for i in range(35) if i % 5 == 0],
              'min_k': [1, 2, 3, 5],
              'sim_options': {'name': ['cosine', 'msd', 'pearson',␣
 ↪'pearson_baseline'], 'user_based': [False]}}

# Performing 3-fold cross validation to tune the hyperparameters
grid_obj = GridSearchCV(KNNBasic, param_grid, measures=['rmse', 'mae'], cv=3,␣
 ↪n_jobs=-1)

# Fitting the data
grid_obj.fit(data)

# Best RMSE score
print(grid_obj.best_score['rmse'])

# Combination of parameters that gave the best RMSE score
print(grid_obj.best_params['rmse'])
```

```
0.9427438084943583
{'k': 30, 'min_k': 3, 'sim_options': {'name': 'msd', 'user_based': False}}
```

This code is similar to the one previously explained. Here is a brief description:

The code sets up a parameter grid to tune hyperparameters for a KNNBasic algorithm with item-based collaborative filtering.

The parameter grid contains different values for the number of neighbors (k) and minimum number of neighbors (min_k), as well as different similarity metrics (cosine, msd, pearson, pearson_baseline).

The code then performs 3-fold cross-validation to tune the hyperparameters and find the combination that results in the best RMSE score.

After fitting the data, the code prints out the best RMSE score and the combination of hyperparameters that resulted in the best score.

Once the **grid search** is complete, we can get the **optimal values for each of those hyperparameters as shown above**

Below we are analysing evaluation metrics - RMSE and MAE at each and every split to analyze the impact of each value of hyperparameters

[ ]:
```
results_df = pd.DataFrame.from_dict(grid_obj.cv_results)
results_df.head()
```

[ ]:

	split0_test_rmse	split1_test_rmse	split2_test_rmse	mean_test_rmse	std_test_rmse	rank_test_rmse	split0_test_mae	split1_test_mae	split2_test_mae	mean_test_mae	std_test_mae	rank_test_mae	mean_fit_time	std_fit_time	mean_test_time	std_test_time	params	param_k	param_min_k
0	1.060798	1.059549	1.053818	1.058055	0.003039	96	0.852163	0.850414	0.846825	0.849801	0.002222	108	8.312761	1.162061	6.689736	0.149401	{'k': 0, 'min_k': 1, 'sim_options': {'name': '…	0	1
1	1.060798	1.059549	1.053818	1.058055	0.003039	89	0.852163	0.850414	0.846825	0.849801	0.002222	95	3.626462	0.396972	7.217132	0.146534			
2	1.060798	1.059549	1.053818	1.058055	0.003039	93	0.852163	0.850414	0.846825	0.849801	0.002222	94	9.545370	1.040723	6.745296	0.228819			
3	1.060798	1.059549	1.053818	1.058055	0.003039	91	0.852163	0.850414	0.846825	0.849801	0.002222	93	7.286118	1.230842	7.868834	0.390965			
4	1.060798	1.059549	1.053818	1.058055	0.003039	95	0.852163	0.850414	0.846825	0.849801	0.002222	107	6.962328	0.504002	7.511113	0.201590			

```
1  {'k': 0, 'min_k': 1, 'sim_options': {'name': '…        0            1
2  {'k': 0, 'min_k': 1, 'sim_options': {'name': '…        0            1
3  {'k': 0, 'min_k': 1, 'sim_options': {'name': '…        0            1
4  {'k': 0, 'min_k': 2, 'sim_options': {'name': '…        0            2

                                  param_sim_options
0           {'name': 'cosine', 'user_based': False}
1              {'name': 'msd', 'user_based': False}
2          {'name': 'pearson', 'user_based': False}
3  {'name': 'pearson_baseline', 'user_based': False}
4           {'name': 'cosine', 'user_based': False}
```

This code creates a pandas DataFrame called results_df from the dictionary returned by the cv_results_ attribute of a GridSearchCV object called grid_obj.

The DataFrame contains information about the results of cross-validation performed on a KNNBasic collaborative filtering algorithm with different hyperparameters.

results_df.head() displays the first 5 rows of the DataFrame, which includes columns such as params (the combination of hyperparameters used), mean_test_rmse (the mean RMSE score for that combination of hyperparameters across all folds of the cross-validation), and rank_test_rmse (the rank of that combination of hyperparameters based on RMSE score).

Now let's build the **final model** by using **tuned values of the hyperparameters** which we received by using grid search cross-validation.

[ ]:
```
# Remove _______ and complete the code
# Creating an instance of KNNBasic with optimal hyperparameter values
similarity_algo_optimized_item = KNNBasic(sim_options={'name': 'msd',␣
 ↪'user_based': False}, k= 30, min_k = 3, verbose=False)

# Training the algorithm on the trainset
similarity_algo_optimized_item.fit(trainset)

# Predicting ratings for the testset
predictions = similarity_algo_optimized_item.test(testset)

# Computing RMSE on testset
accuracy.rmse(predictions)
```

```
RMSE: 0.9468
```

[ ]:
```
0.9468088882738848
```

This code is a machine learning code written in Python using the Surprise library. Here is a description of each line:

similarity_algo_optimized_item = KNNBasic(sim_options={'name': 'msd', 'user_based': False}, k= 30, min_k = 3, verbose=False): An instance of the KNNBasic algorithm is created with the optimal hyperparameter values.

The sim_options parameter is set to use the 'msd' similarity metric and item-based collaborative filtering. The k parameter is set to 30, which represents the number of similar items to consider.

The min_k parameter is set to 3, which is the minimum number of similar items required to make a prediction. The verbose parameter is set to False, which means no additional information will be displayed during the algorithm's execution.

similarity_algo_optimized_item.fit(trainset): The algorithm is trained on the training set.

predictions = similarity_algo_optimized_item.test(testset): Predictions are made on the test set using the trained model.

accuracy.rmse(predictions): The root mean squared error (RMSE) is computed on the test set to evaluate the performance of the model. The rmse method is called from the accuracy module of the Surprise library.

Write your Answer here: RMSE for tuned item based collaborative filtering recommendation system is 0.9468.

1.7.4 Q 4.4 What is the Predicted rating for an item with userId =4 and for movieId= 10 and movieId=3 using tuned item based collaborative filtering? (1 Mark)

Let's us now predict rating for an user with `userId=4` and for `movieId=10` with the optimized model as shown below

[ ]:
```
# Remove _______ and complete the code
similarity_algo_optimized_item.predict(uid = 4, iid = 10, r_ui=3, verbose=True)
```

```
user: 4          item: 10         r_ui = 3.00   est = 4.30   {'actual_k': 30,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=10, r_ui=4, est=4.298279280483517, details={'actual_k':
30, 'was_impossible': False})
```

Write your Answer here: Predicted rating for an item with userId =4 and for movieId= 10 is 4.30.

Let's predict the rating for the same `userId=4` but for a movie which this user has not interacted before i.e. `movieId=3`, by using the optimized model:

[ ]:
```
# Remove _______ and complete the code
similarity_algo_optimized_item.predict(uid = 4, iid = 3, verbose=True)
```

```
user: 4          item: 3          r_ui = None   est = 3.86   {'actual_k': 30,
'was_impossible': False}
```

[ ]:
```
Prediction(uid=4, iid=3, r_ui=None, est=3.859023126306401, details={'actual_k':
30, 'was_impossible': False})
```

Write your Answer here: Predicted rating for an item with userId =4 and for movieId= 4 is 3.86.

1.7.5 Identifying similar items to a given item (nearest neighbors)

We can also find out the similar items to a given item or its nearest neighbors based on this KNNBasic algorithm. Below we are finding 5 most similar items to the `movieId=3` based on the `msd` distance metric

[ ]:
```
# Remove _______ and complete the code
similarity_algo_optimized_item.get_neighbors(3 , k=5)
```

[ ]:
```
[31, 37, 42, 48, 73]
```

Predicted top 5 movies for userId=4 with similarity based recommendation system

[ ]:
```
# Remove _______ and complete the code
recommendations = get_recommendations(rating, 4, 5,␣
↪similarity_algo_optimized_item)
```

1.7.6 Q 4.5 Predict the top 5 movies for userId=4 with similarity based recommendation system (1 Mark)

[ ]:
```
recommendations
```

[ ]:
```
[(84, 5), (1040, 5), (2481, 5), (3078, 5), (3116, 5)]
```

1.8 Model 4: Based Collaborative Filtering - Matrix Factorization using SVD (7 Marks)

Model-based Collaborative Filtering is a **personalized recommendation system**, the recommendations are based on the past behavior of the user and it is not dependent on any additional information. We use **latent features** to find recommendations for each user.

Latent Features: The features that are not present in the empirical data but can be inferred from the data. For example:

User	Movie	Rating
Ram	Rambo	8
Hari	The Notebook	7
Rahi	The Usual Suspects	8
Rahim	Dumb and Dumber	7

Now if we notice the above movies closely:

	Action	Romance	Suspense	Comedy
Rambo	Yes	No	No	No
The Notebook	No	Yes	No	No
The Usual Suspects	Yes	No	Yes	No
Dumb and Dumber	No	No	No	Yes

Here **Action**, **Romance**, **Suspense** and **Comedy** are latent features of the corresponding movies. Similarly, we can compute the latent features for users as shown below:

	Action	Romance	Suspense	Comedy
Ram	Yes	No	No	No
Hari	No	Yes	No	No
Rahi	Yes	No	Yes	No
Rahim	No	No	No	Yes

1.8.1 Singular Value Decomposition (SVD)

SVD is used to **compute the latent features** from the **user-item matrix**. But SVD does not work when we **miss values** in the **user-item matrix**.

First we need to convert the below movie-rating dataset:

User	Movie	Rating
Ram	Rambo	8
Hari	The Notebook	7
Rahi	The Usual Suspects	8
Rahim	Dumb and Dumber	7

into an user-item matrix as shown below:

	Rambo	The Notebook	The Usual Suspects	Dumb and Dumber
Ram	8	0	0	0
Hari	0	7	0	0
Rahi	0	0	8	0
Rahim	0	0	0	7

We have already done this above while computing cosine similarities.

SVD decomposes this above matrix into three separate matrices: - U matrix - Sigma matrix - V transpose matrix

	Action	Romance	Suspense	Comedy
Ram	8	0	2	0
Hari	0	9	1	2
Rahi	7	1	8	1
Rahim	0	2	0	8

U-matrix the above matrix is a n x k matrix, where: - n is number of users - k is number of latent features

	Action	Romance	Suspense	Comedy
Action	20	0	0	0
Romance	0	15	0	0
Suspense	0	0	30	0
Comdey	0	0	0	25

Sigma-matrix the above matrix is a k x k matrix, where: - k is number of latent features - Each diagonal entry is the singular value of the original interaction matrix

	Rambo	The Notebook	The Usual Suspects	Dumb and Dumber
Action	8	0	5	0
Romance	0	7	2	5
Suspense	2	0	8	0
Comedy	0	1	1	8

V-transpose matrix the above matrix is a kxn matrix, where: - k is the number of latent features - n is the number of items

1.8.2 Build a baseline matrix factorization recommendation system

[ ]:
```
# Remove _______ and complete the code

# Using SVD matrix factorization
algo_svd = SVD()

# Training the algorithm on the trainset
algo_svd.fit(trainset)

# Predicting ratings for the testset
predictions = algo_svd.test(testset)

# Computing RMSE on the testset
accuracy.rmse(predictions)
```

```
RMSE: 0.9017
```

[ ]:
```
0.9017495272218696
```

This code is using the SVD (Singular Value Decomposition) matrix factorization algorithm to predict ratings on the testset.

algo_svd = SVD() creates an instance of the SVD algorithm.

algo_svd.fit(trainset) trains the algorithm on the trainset.

predictions = algo_svd.test(testset) generates predictions for the testset using the trained algorithm.

accuracy.rmse(predictions) computes and prints the RMSE (Root Mean Squared Error) between the predicted ratings and the actual ratings in the testset.

1.8.3 Q 5.1 What is the RMSE for baseline SVD based collaborative filtering recommendation system? (1 Mark)

Write your Answer here: RMSE for baseline SVD based collaborative filtering recommendation system is 0.9017.

1.8.4 Q 5.2 What is the Predicted rating for an user with userId =4 and for movieId= 10 and movieId=3? (1 Mark)

Let's us now predict rating for an user with `userId=4` and for `movieId=10`

[ ]:
```
# Remove _______ and complete the code
algo_svd.predict(uid = 4, iid = 10, r_ui=3, verbose=True)
```

```
user: 4          item: 10         r_ui = 3.00   est = 4.22   {'was_impossible':
False}
```

[ ]:
```
Prediction(uid=4, iid=10, r_ui=4, est=4.22422859715209,
details={'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 10 is 4.22.

Let's predict the rating for the same `userId=4` but for a movie which this user has not interacted before i.e. `movieId=3`:

[ ]:
```
# Remove _______ and complete the code
algo_svd.predict(4, 3, verbose=True)
```

```
user: 4          item: 3          r_ui = None   est = 3.70   {'was_impossible':
False}
```

[ ]:
```
Prediction(uid=4, iid=3, r_ui=None, est=3.698749994379488,
details={'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 4 is 3.70.

1.8.5 Improving matrix factorization based recommendation system by tuning its hyper-parameters

In SVD, rating is predicted as -

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

If user u is unknown, then the bias b_u and the factors p_u are assumed to be zero. The same applies for item i with b_i and q_i.

To estimate all the unknown, we minimize the following regularized squared error:

$$\sum_{r_{ui} \in R_{train}} \left(r_{ui} - \hat{r}_{ui}\right)^2 + \lambda \left(b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2\right)$$

The minimization is performed by a very straightforward **stochastic gradient descent**:

$$\begin{aligned} b_u &\leftarrow b_u + \gamma \left(e_{ui} - \lambda b_u\right) \\ b_i &\leftarrow b_i + \gamma \left(e_{ui} - \lambda b_i\right) \\ p_u &\leftarrow p_u + \gamma \left(e_{ui} \cdot q_i - \lambda p_u\right) \\ q_i &\leftarrow q_i + \gamma \left(e_{ui} \cdot p_u - \lambda q_i\right) \end{aligned}$$

There are many hyperparameters to tune in this algorithm, you can find a full list of hyperparameters here

Below we will be tuning only three hyperparameters - - **n_epochs**: The number of iteration of the SGD algorithm - **lr_all**: The learning rate for all parameters - **reg_all**: The regularization term for all parameters

1.8.6 Q 5.3 Perform hyperparameter tuning for the baseline SVD based collaborative filtering recommendation system and find the RMSE for tuned SVD based collaborative filtering recommendation system? (3 Marks)

[ ]:
```
# Remove _______ and complete the code

# Set the parameter space to tune
param_grid = {'n_epochs': [10, 20, 30], 'lr_all': [0.001, 0.005, 0.01],
              'reg_all': [0.2, 0.4, 0.6]}

# Performing 3-fold gridsearch cross validation
gs = GridSearchCV(SVD, param_grid, measures=['rmse', 'mae'], cv=3, n_jobs=-1)

# Fitting data
gs.fit(data)

# Best RMSE score
print(gs.best_score['rmse'])

# Combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

```
0.8943769425860374
{'n_epochs': 30, 'lr_all': 0.01, 'reg_all': 0.2}
```

This code performs hyperparameter tuning for the SVD (singular value decomposition) algorithm

using grid search cross-validation.

First, a parameter grid is defined with various values for the hyperparameters n_epochs (number of epochs for the SGD optimization algorithm), lr_all (learning rate for all parameters), and reg_all (regularization term for all parameters).

Next, the GridSearchCV function from the surprise library is used to perform 3-fold grid search cross-validation, optimizing for the RMSE and MAE metrics. The n_jobs argument is set to -1 to use all available CPU cores for parallel processing.

After fitting the data, the best RMSE score is printed using the best_score attribute of the GridSearchCV object, and the combination of hyperparameters that gave the best RMSE score is printed using the best_params attribute of the GridSearchCV object.

Once the **grid search** is complete, we can get the **optimal values** for each of those hyperparameters, as shown above.

Below we are analysing evaluation metrics - RMSE and MAE at each and every split to analyze the impact of each value of hyperparameters

[ ]:
```
results_df = pd.DataFrame.from_dict(gs.cv_results)
results_df.head()
```

[ ]:

	split0_test_rmse	split1_test_rmse	split2_test_rmse	mean_test_rmse	\
0	0.942356	0.946055	0.942223	0.943545	
1	0.946654	0.950534	0.945847	0.947678	
2	0.951563	0.956048	0.950923	0.952845	
3	0.906650	0.910691	0.904948	0.907430	
4	0.913911	0.917619	0.911785	0.914438	

	std_test_rmse	rank_test_rmse	split0_test_mae	split1_test_mae	\
0	0.001776	25	0.737316	0.739553	
1	0.002046	26	0.742564	0.744417	
2	0.002280	27	0.748544	0.750691	
3	0.002408	11	0.701998	0.704967	
4	0.002411	15	0.710232	0.712337	

	split2_test_mae	mean_test_mae	std_test_mae	rank_test_mae	mean_fit_time	\
0	0.738325	0.738398	0.000915	25	0.835168	
1	0.742289	0.743090	0.000945	26	0.834211	
2	0.748239	0.749158	0.001091	27	0.870453	
3	0.702085	0.703016	0.001380	9	0.794008	
4	0.709263	0.710611	0.001283	15	0.861538	

	std_fit_time	mean_test_time	std_test_time	\
0	0.053319	0.499357	0.014224	
1	0.024491	0.507418	0.021669	
2	0.052825	0.494645	0.005512	
3	0.032267	0.497058	0.028663	
4	0.028507	0.468000	0.036547	

```
                                                params  param_n_epochs  \
0  {'n_epochs': 10, 'lr_all': 0.001, 'reg_all': 0.2}              10
1  {'n_epochs': 10, 'lr_all': 0.001, 'reg_all': 0.4}              10
2  {'n_epochs': 10, 'lr_all': 0.001, 'reg_all': 0.6}              10
3  {'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.2}              10
4  {'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.4}              10

   param_lr_all  param_reg_all
0         0.001            0.2
1         0.001            0.4
2         0.001            0.6
3         0.005            0.2
4         0.005            0.4
```

Now, we will **the build final model** by using **tuned values** of the hyperparameters, which we received using grid search cross-validation above.

[ ]:
```
# Remove _______ and complete the code

# Building the optimized SVD model using optimal hyperparameter search
svd_algo_optimized = SVD(n_epochs= 30, lr_all= 0.01, reg_all=0.2)

# Training the algorithm on the trainset
svd_algo_optimized.fit(trainset)

# Predicting ratings for the testset
predictions = svd_algo_optimized.test(testset)

# Computing RMSE
accuracy.rmse(predictions)
```

```
RMSE: 0.8952
```

[ ]:
```
0.8952324649512264
```

1.8.7 Q 5.4 What is the Predicted rating for an user with userId =4 and for movieId= 10 and movieId=3 using SVD based collaborative filtering? (1 Mark)

Let's us now predict rating for an user with `userId=4` and for `movieId=10` with the optimized model

[ ]:
```
# Remove _______ and complete the code
svd_algo_optimized.predict(4, 10, r_ui = 3, verbose=True)
```

```
user: 4          item: 10         r_ui = 3.00   est = 3.99   {'was_impossible':
False}
```

[ ]:
```
Prediction(uid=4, iid=10, r_ui=4, est=3.9901619659500946,
details={'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 10 is 3.99.

Let's predict the rating for the same `userId=4` but for a movie which this user has not interacted before i.e. `movieId=3`:

[ ]:
```
# Remove _______ and complete the code
svd_algo_optimized.predict(4, 3, verbose=True)
```

```
user: 4          item: 3          r_ui = None   est = 3.64   {'was_impossible':
False}
```

[ ]:
```
Prediction(uid=4, iid=3, r_ui=None, est=3.6402055706052314,
details={'was_impossible': False})
```

Write your Answer here: Predicted rating for an user with userId =4 and for movieId= 4 is 3.64.

1.8.8 Q 5.5 Predict the top 5 movies for userId=4 with SVD based recommendation system?(1 Mark)

[ ]:
```
# Remove _______ and complete the code
get_recommendations(rating, 4, 5, svd_algo_optimized)
```

[ ]:
```
[(116, 4.9913802447237465),
 (1192, 4.972874528318438),
 (3310, 4.970551037774074),
 (926, 4.94459271315041),
 (5114, 4.929828152928015)]
```

1.8.9 Predicting ratings for already interacted movies

Below we are comparing the rating predictions of users for those movies which has been already watched by an user. This will help us to understand how well are predictions are as compared to the actual ratings provided by users

[ ]:
```
def predict_already_interacted_ratings(data, user_id, algo):

    # Creating an empty list to store the recommended movie ids
    recommendations = []

    # Creating an user item interactions matrix
    user_item_interactions_matrix = data.pivot(index='userId',␣
 ↪columns='movieId', values='rating')

    # Extracting those movie ids which the user_id has interacted already
```

```
    interacted_movies = user_item_interactions_matrix.
↪loc[user_id][user_item_interactions_matrix.loc[user_id].notnull()].index.
↪tolist()

    # Looping through each of the movie id which user_id has interacted already
    for item_id in interacted_movies:

        # Extracting actual ratings
        actual_rating = user_item_interactions_matrix.loc[user_id, item_id]

        # Predicting the ratings for those non interacted movie ids by this user
        predicted_rating = algo.predict(user_id, item_id).est

        # Appending the predicted ratings
        recommendations.append((item_id, actual_rating, predicted_rating))

    # Sorting the predicted ratings in descending order
    recommendations.sort(key=lambda x: x[1], reverse=True)

    return pd.DataFrame(recommendations, columns=['movieId', 'actual_rating',␣
↪'predicted_rating']) # returing top n highest predicted rating movies for␣
↪this user
```

This code defines a function predict_already_interacted_ratings that takes three inputs:

data - a pandas DataFrame containing userId, movieId, and rating columns user_id - an integer representing the ID of the user for whom recommendations are being generated algo - an instance of a prediction algorithm that has been trained on the data DataFrame The function aims to recommend movies to the given user_id that they haven't interacted with previously, i.e., haven't rated.

Here's how the function works:

An empty list recommendations is created to store the recommended movie IDs. A user-item interaction matrix is created from the data DataFrame.

A list interacted_movies is created containing the movie IDs that the user_id has already interacted with, i.e., rated.

A for loop is started to iterate through each movie ID that the user_id has already interacted with.

Inside the loop, the actual rating that the user_id gave to that particular movie is extracted from the user-item interaction matrix.

The prediction algorithm algo is used to predict the rating that the user_id would have given to the movie.

The predicted rating and actual rating are both appended to the recommendations list.

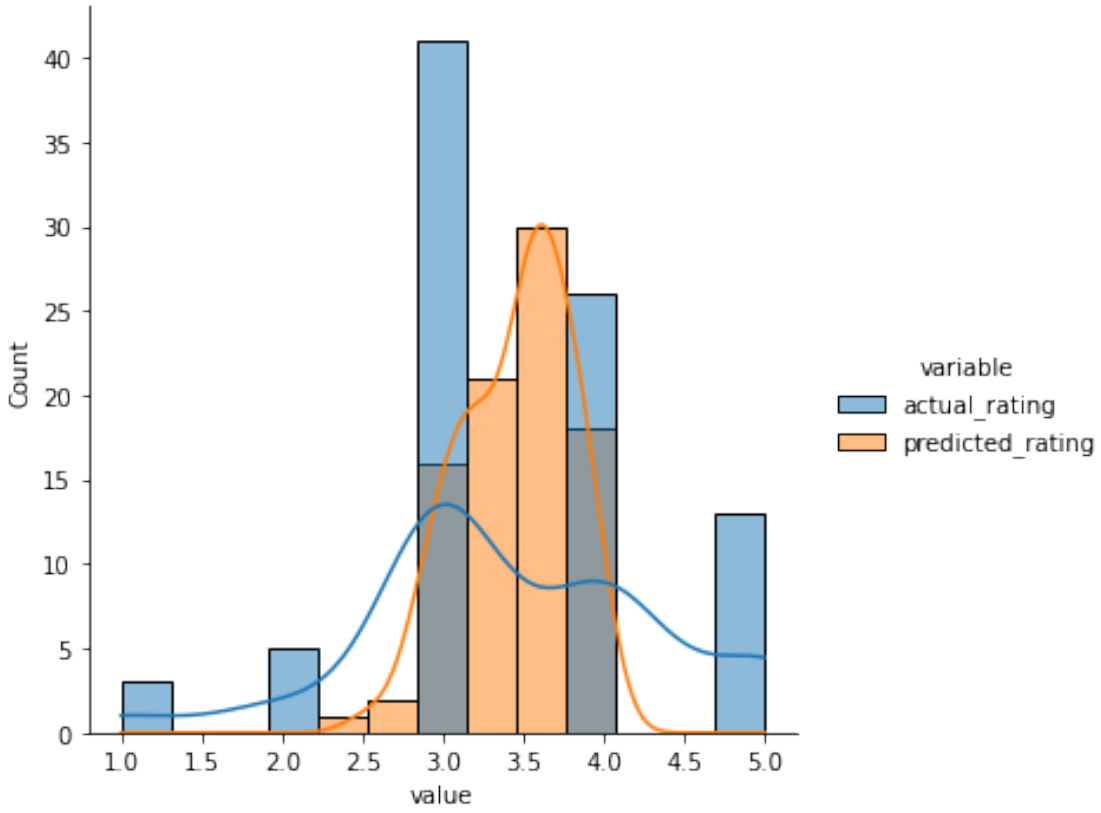
The recommendations list is sorted in descending order based on the predicted ratings.

A pandas DataFrame is returned containing the movie IDs, actual ratings, and predicted ratings, sorted by the highest predicted rating first.

In summary, this function is used to generate personalized movie recommendations for a given user based on the movies they have already interacted with, using a prediction algorithm that has been trained on the data.

Here we are comparing the predicted ratings by **similarity based recommendation** system against actual ratings for **userId=7**

[ ]:
```
predicted_ratings_for_interacted_movies =␣
↪predict_already_interacted_ratings(rating, 7, similarity_algo_optimized_item)
df = predicted_ratings_for_interacted_movies.melt(id_vars='movieId',␣
↪value_vars=['actual_rating', 'predicted_rating'])
sns.displot(data=df, x='value', hue='variable', kde=True);
```



Write your Answer here: We are getting the most predicted values in the range of 3 to 4. The number of actual values is high for the ratings 3.0 and 4.0. There are more than 10 actual ratings between the range of 4.5 and 5, but there are no or very less predicted ratings between that range.

The code performs the following tasks:

It calls the function 'predict_already_interacted_ratings' with the input parameters as rating data, user_id 7 and a pre-trained instance of 'KNNBasic' algorithm called 'similarity_algo_optimized_item'.

The function returns a dataframe containing the actual ratings and predicted ratings for the movies already interacted by the user with id 7.

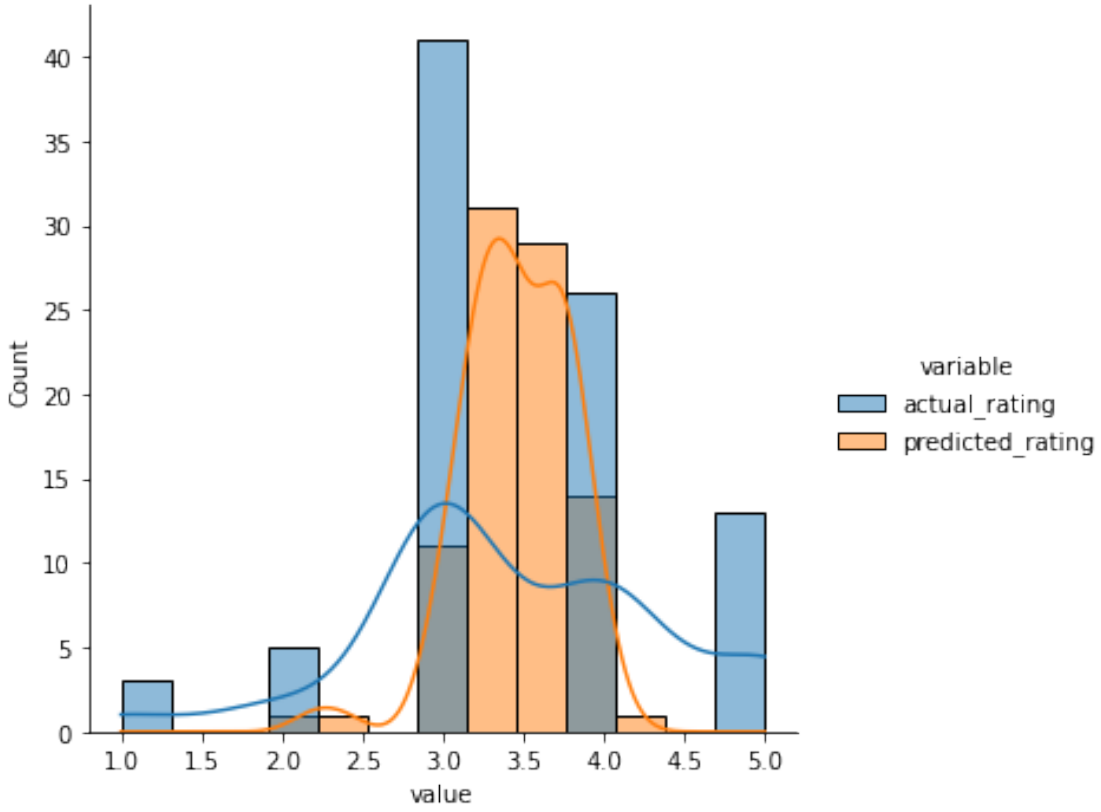
The code then uses the 'melt' function from the pandas library to convert the dataframe from wide format to long format, with 'movieId' as the id variable and 'actual_rating' and 'predicted_rating' as the value variables.

Finally, the code uses the 'displot' function from the seaborn library to create a visualization of the distribution of actual and predicted ratings for the movies already interacted by user 7.

The resulting plot is a histogram with a KDE (kernel density estimate) curve, with actual and predicted ratings shown in different colors.

Below we are comparing the predicted ratings by `matrix factorization based recommendation` system against actual ratings for `userId=7`

[ ]:
```
predicted_ratings_for_interacted_movies =␣
 ↪predict_already_interacted_ratings(rating, 7, svd_algo_optimized)
df = predicted_ratings_for_interacted_movies.melt(id_vars='movieId',␣
 ↪value_vars=['actual_rating', 'predicted_rating'])
sns.displot(data=df, x='value', hue='variable', kde=True);
```



```
[ ]: # Instantiating Reader scale with expected rating scale
reader = Reader(rating_scale=(0, 5))

# Loading the rating dataset
data = Dataset.load_from_df(rating[['userId', 'movieId', 'rating']], reader)

# Splitting the data into train and test dataset
trainset, testset = train_test_split(data, test_size=0.2, random_state=42)
```

This code is related to the data preparation process for building a recommendation system. Here is what each line does:

Reader(rating_scale=(0, 5)): Create an instance of the Reader class from Surprise library and set the rating scale from 0 to 5. This tells Surprise how the ratings are scaled in the dataset.

Dataset.load_from_df(rating[['userId', 'movieId', 'rating']], reader): Load the ratings dataset from pandas DataFrame and use the reader object created earlier to parse the data. This will create an object of the Dataset class from Surprise.

train_test_split(data, test_size=0.2, random_state=42): Split the data object into training and testing sets with a test size of 0.2, meaning that 20% of the data will be used for testing.

The random state parameter is set to 42 to ensure that the same random data split can be reproduced. This function returns two objects: trainset and testset, which will be used in training and testing the recommendation algorithm, respectively.

1.9 Precision and Recall @ k

RMSE is not the only metric we can use here. We can also examine two fundamental measures, precision and recall. We also add a parameter k which is helpful in understanding problems with multiple rating outputs.

Precision@k - It is the **fraction of recommended items that are relevant in `top k` predictions**. Value of k is the number of recommendations to be provided to the user. One can choose a variable number of recommendations to be given to a unique user.

Recall@k - It is the **fraction of relevant items that are recommended to the user in `top k` predictions**.

Recall - It is the **fraction of actually relevant items that are recommended to the user** i.e. if out of 10 relevant movies, 6 are recommended to the user then recall is 0.60. Higher the value of recall better is the model. It is one of the metrics to do the performance assessment of classification models.

Precision - It is the **fraction of recommended items that are relevant actually** i.e. if out of 10 recommended items, 6 are found relevant by the user then precision is 0.60. The higher the value of precision better is the model. It is one of the metrics to do the performance assessment of classification models.

See the Precision and Recall @ k section of your notebook and follow the instructions to compute various precision/recall values at various values of k.

To know more about precision recall in Recommendation systems refer to these links :

https://surprise.readthedocs.io/en/stable/FAQ.html

https://medium.com/@m__n__malaeb/recall-and-precision-at-k-for-recommender-systems-618483226c54

1.9.1 Question6: Compute the precision and recall, for each of the 6 models, at k = 5 and 10. This is 6 x 2 = 12 numerical values? (4 marks)

[ ]:
```python
# Function can be found on surprise documentation FAQs
def precision_recall_at_k(predictions, k=10, threshold=3.5):
    """Return precision and recall at k metrics for each user"""

    # First map the predictions to each user.
    user_est_true = defaultdict(list)
    for uid, _, true_r, est, _ in predictions:
        user_est_true[uid].append((est, true_r))

    precisions = dict()
    recalls = dict()
    for uid, user_ratings in user_est_true.items():

        # Sort user ratings by estimated value
        user_ratings.sort(key=lambda x: x[0], reverse=True)

        # Number of relevant items
        n_rel = sum((true_r >= threshold) for (_, true_r) in user_ratings)

        # Number of recommended items in top k
        n_rec_k = sum((est >= threshold) for (est, _) in user_ratings[:k])

        # Number of relevant and recommended items in top k
        n_rel_and_rec_k = sum(((true_r >= threshold) and (est >= threshold))
                              for (est, true_r) in user_ratings[:k])

        # Precision@K: Proportion of recommended items that are relevant
        # When n_rec_k is 0, Precision is undefined. We here set it to 0.

        precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 0

        # Recall@K: Proportion of relevant items that are recommended
        # When n_rel is 0, Recall is undefined. We here set it to 0.

        recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 0

    return precisions, recalls
```

This code defines a function precision_recall_at_k which takes in the predictions, k (the number of recommended items), and threshold (the minimum rating considered as relevant). The function

calculates the precision and recall at k metrics for each user in the predictions.

First, the function creates a dictionary user_est_true where each key represents a user and the value is a list of tuples containing the estimated rating and the true rating for each movie. Then, the function sorts the ratings by estimated value in descending order.

For each user, the function calculates the number of relevant items (those with a true rating greater than or equal to the threshold), the number of recommended items in the top k, and the number of relevant and recommended items in the top k.

Using these values, the function calculates the precision at k as the proportion of recommended items that are relevant and the recall at k as the proportion of relevant items that are recommended. If there are no relevant items or recommended items in the top k, the precision or recall is set to 0.

Finally, the function returns two dictionaries precisions and recalls which contain the precision and recall at k metrics for each user.

[ ]:
```
# A basic cross-validation iterator.
kf = KFold(n_splits=5)

# Make list of k values
K = [5, 10]

# Remove _______ and complete the code
# Make list of models
models = [algo_knn_user, similarity_algo_optimized_user, \
          algo_knn_item, similarity_algo_optimized_item, \
          algo_svd, svd_algo_optimized]

for k in K:
    for model in models:
        print('> k={}, model={}'.format(k,model.__class__.__name__))
        p = []
        r = []
        for trainset, testset in kf.split(data):
            model.fit(trainset)
            predictions = model.test(testset, verbose=False)
            precisions, recalls = precision_recall_at_k(predictions, k=k,␣
 ↪threshold=3.5)

            # Precision and recall can then be averaged over all users
            p.append(sum(prec for prec in precisions.values()) /␣
 ↪len(precisions))
            r.append(sum(rec for rec in recalls.values()) / len(recalls))

        print('-----> Precision: ', round(sum(p) / len(p), 3))
        print('-----> Recall: ', round(sum(r) / len(r), 3))
```

```
> k=5, model=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.
-----> Precision:  0.764
-----> Recall:  0.409
> k=5, model=KNNBasic
-----> Precision:  0.771
-----> Recall:  0.416
> k=5, model=KNNBasic
-----> Precision:  0.611
-----> Recall:  0.331
> k=5, model=KNNBasic
-----> Precision:  0.683
-----> Recall:  0.359
> k=5, model=SVD
-----> Precision:  0.749
-----> Recall:  0.378
> k=5, model=SVD
-----> Precision:  0.747
-----> Recall:  0.387
> k=10, model=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.
-----> Precision:  0.749
-----> Recall:  0.55
> k=10, model=KNNBasic
-----> Precision:  0.754
-----> Recall:  0.559
> k=10, model=KNNBasic
-----> Precision:  0.597
-----> Recall:  0.477
> k=10, model=KNNBasic
-----> Precision:  0.659
-----> Recall:  0.504
```

```
> k=10, model=SVD
-----> Precision:  0.733
-----> Recall:  0.518
> k=10, model=SVD
-----> Precision:  0.73
-----> Recall:  0.524
```

This code is performing a cross-validation procedure to evaluate the performance of different recommendation models.

The KFold function from scikit-learn is used to create a basic cross-validation iterator that splits the dataset into 5 folds.

The K variable is a list of values for k parameter, which is the number of top items to recommend.

The models list contains different recommendation models that are to be evaluated.

The for loop iterates over each k value and each model, and for each combination, it fits the model on the training set and tests it on the test set. The precision_recall_at_k function is used to calculate the precision and recall metrics for each user in the test set. The precision and recall values are then averaged over all users in the test set.

Finally, the code prints the precision and recall metrics for each combination of k and model.

1.9.2 Question 7 (5 Marks)

7.1 Compare the results from the base line user-user and item-item based models.

7.2 How do these baseline models compare to each other with respect to the tuned user-user and item-item models?

7.3 The matrix factorization model is different from the collaborative filtering models. Briefly describe this difference. Also, compare the RMSE and precision recall for the models.

7.4 Does it improve? Can you offer any reasoning as to why that might be?

Write your Answer here:_____

Answer 7.1

K	Model	Precision	Recall
5	algo_knn_user	0.764	0.409
5	similarity_algo_optimized_user	0.771	0.416
5	algo_knn_item	0.611	0.331
5	similarity_algo_optimized_item	0.683	0.359
5	algo_svd	0.749	0.378
5	svd_algo_optimized	0.747	0.387

K	Model	Precision	Recall
10	algo_knn_user	0.749	0.55
10	similarity_algo_optimized_user	0.754	0.559
10	algo_knn_item	0.597	0.477
10	similarity_algo_optimized_item	0.659	0.504
10	algo_svd	0.733	0.518
10	svd_algo_optimized	0.73	0.524

Model	RMSE
algo_knn_user	0.9925
similarity_algo_optimized_user	0.9571
algo_knn_item	1.0032
similarity_algo_optimized_item	0.9468
algo_svd	0.9017
svd_algo_optimized	0.8952

Before optimization, baseline user-user based model has performed better than baseline item-item based model, but after the optimization, baseline item-item based model has performed better than user-user based model.

Answer 7.2

The tuned user-user and item-item models have performed better than the baseline models as the precision and the recall is increased, and the RMSE is decreased after optimization.

Answer 7.3

In collaborative filtering, the past data, of the users, is used to analyse the behavior of other users with similar likings.

Matrix factorization is used to decompose the original matrix, say user-item matrix, into two matrices in such a way that their dot product will result in a matrix that is similar to the original matrix.

According to the table mentioned in answer 7.1, the RMSE of matrix factorization models is better than collaborative filtering models, but the precision and the recall values of collaborative filtering models are better than the matrix factorization model.

Answer 7.4

The RMSE score of the matrix factorization model is better. The collaborative models do not perform well because of the ample amount of data and the sparsity of the rating matrix. Matrix factorization is used to tackle these issues.

It decomposes the original matrix into two matrices, and it creates the latent factors, and then it maps the users and the items against those latent factors.

1.9.3 Conclusions

In this case study, we saw three different ways of building recommendation systems: - rank-based using averages - similarity-based collaborative filtering - model-based (matrix factorization) collaborative filtering

We also understood advantages/disadvantages of these recommendation systems and when to use which kind of recommendation systems. Once we build these recommendation systems, we can use **A/B Testing** to measure the effectiveness of these systems.

Here is an article explaining how Amazon use **A/B Testing** to measure effectiveness of its recommendation systems.