Zee Recommender Systems

November 20, 2022

0.0.1 Problem Statement

We have been given three datasets - Ratings - Users - Movies

and using above three we have to create a Recommender System to show personalized movie recommendations based on ratings given by a user and other users similar to them in order to improve user experience.

[1]:
```
# Loading all the necessary libraries

# for multidimensional array processing
import numpy as np

# for working with structured dataset
import pandas as pd
pd.options.display.float_format = "{:.2f}".format

# for basic plotting functionalities
import matplotlib.pyplot as plt

# for plotting advanced graphs
import seaborn as sns

# to normalize data for getting best linear regression model
from sklearn.preprocessing import StandardScaler

# for using KNN imputation
from sklearn.impute import KNNImputer

# for performing datetime related operations
from datetime import datetime

# to see progress of operation
from tqdm.notebook import tqdm_notebook

# for using pearson correlation
from scipy.stats import pearsonr

# importing nearest neighbor for finding similarity
```

```
from sklearn.neighbors import NearestNeighbors

# for sparse CSR representation
from scipy import sparse

# for using matrix factorization
from cmfrec import CMF

# importing mean_squared_error and mean_absolute_percentage_error
from sklearn.metrics import mean_absolute_percentage_error, mean_squared_error

# importing k means clustering to visualize MF (matrix factorization) results
from sklearn.cluster import KMeans

# to suppress any warnings coming out
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
ratings = pd.read_csv('zee-ratings.dat', encoding='ISO-8859-1', delimiter = "::
  ↪")
ratings.head()
```

[2]:

	UserID	MovieID	Rating	Timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

[3]:
```
users = pd.read_csv('zee-users.dat', encoding='ISO-8859-1', delimiter = "::")
users.head()
```

[3]:

	UserID	Gender	Age	Occupation	Zip-code
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455

[4]:
```
movies = pd.read_csv('zee-movies.dat', encoding='ISO-8859-1', delimiter = "::")
movies.head()
```

[4]:

	Movie ID	Title	Genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama

```
4         5  Father of the Bride Part II (1995)                  Comedy
```

0.0.2 1. Exploratory Data Analysis & Analyzing Basic Metrics

[5]:
```
ratings.shape
```

[5]:
```
(1000209, 4)
```

On expecting the shape of the dataframe we can see that there are 1000209 rows and 4 columns. Hence, we can say that we are working with alarge amount of ratings data.

Attribute Information of the Ratings Data - UserIDs :- range between 1 and 6040 - MovieIDs :- range between 1 and 3952 - Ratings :- are made on a 5-star scale (whole-star ratings only) - Timestamp :- is represented in seconds

[6]:
```
users.shape
```

[6]:
```
(6040, 5)
```

On expecting the shape of the dataframe we can see that there are 6404 rows and 5 columns. Hence, we can say that we have good amount of users in our database

Attribute Information of the Users Data - UserIDs :- range between 1 and 6040 - Gender :- is denoted by a “M” for male and “F” for female - Age :- is chosen from the following ranges: 1: “Under 18” 18: “18-24” 25: “25-34” 35: “35-44” 45: “45-49” 50: “50-55” 56: “56+” - Occupation :- is chosen from the following choices: 0: “other” or not specified 1: “academic/educator” 2: “artist” 3: “clerical/admin” 4: “college/grad student” 5: “customer service” 6: “doctor/health care” 7: “executive/managerial” 8: “farmer” 9: “homemaker” 10: “K-12 student” 11: “lawyer” 12: “programmer” 13: “retired” 14: “sales/marketing” 15: “scientist” 16: “self-employed” 17: “technician/engineer” 18: “tradesman/craftsman” 19: “unemployed” 20: “writer”

[7]:
```
movies.shape
```

[7]:
```
(3883, 3)
```

On expecting the shape of the dataframe we can see that there are 6404 rows and 5 columns. Hence, we can say that we have good amount of users in our database

Attribute Information of the Movies Data - MovieIDs :- range between 1 and 3952 - Titles :- are identical to titles provided by the IMDB (including year of release) - Genres :- are pipe-separated and are selected from the following genres: Action Adventure Animation Children’s Comedy Crime Documentary Drama Fantasy Film-Noir Horror Musical Mystery Romance Sci-Fi Thriller War Western

[8]:
```
ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000209 entries, 0 to 1000208
Data columns (total 4 columns):
 #   Column     Non-Null Count    Dtype
```

```
---  ------     --------------    -----
 0   UserID     1000209 non-null  int64
 1   MovieID    1000209 non-null  int64
 2   Rating     1000209 non-null  int64
 3   Timestamp  1000209 non-null  int64
dtypes: int64(4)
memory usage: 30.5 MB
```

On checking the information of ratings data using info() method we can see that there no null values present in the dataset.

Also, all columns present here are integer data types i.e. int64

[9]:
```
users.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6040 entries, 0 to 6039
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   UserID      6040 non-null   int64
 1   Gender      6040 non-null   object
 2   Age         6040 non-null   int64
 3   Occupation  6040 non-null   int64
 4   Zip-code    6040 non-null   object
dtypes: int64(3), object(2)
memory usage: 236.1+ KB
```

On checking the information of users data using info() method we can see that there no null values present in the dataset.

Also, all columns present here are integer data types i.e. int64 except Gender and Zip-Code which are of object data type i.e. indicates string values are present.

[10]:
```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3883 entries, 0 to 3882
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Movie ID  3883 non-null   int64
 1   Title     3883 non-null   object
 2   Genres    3883 non-null   object
dtypes: int64(1), object(2)
memory usage: 91.1+ KB
```

On checking the information of movies data using info() method we can see that there no null values present in the dataset.

Also, Movie ID is of integer data types i.e. int64 while, Title and Genres are of object data type

i.e. indicates string values are present.

Range of Columns

[11]:
```
i = 1

for col in ratings.columns:
    if ratings[col].dtype == 'object':
        print(str(i) + '. "' + col + '" is categorical column with total unique␣
 ↪values as: ' + str(len(ratings[col].unique())))
    else:
        print(str(i) + '. "' + col + '" is continuous column with min, max and␣
 ↪mean values as: ', ratings[col].min(), ratings[col].max(), ratings[col].
 ↪mean())
    i += 1
```

```
1. "UserID" is continuous column with min, max and mean values as:  1 6040
3024.512347919285
2. "MovieID" is continuous column with min, max and mean values as:  1 3952
1865.5398981612843
3. "Rating" is continuous column with min, max and mean values as:  1 5
3.581564453029317
4. "Timestamp" is continuous column with min, max and mean values as:  956703932
1046454590 972243695.4046655
```

[12]:
```
ratings.describe()
```

[12]:

	UserID	MovieID	Rating	Timestamp
count	1000209.00	1000209.00	1000209.00	1000209.00
mean	3024.51	1865.54	3.58	972243695.40
std	1728.41	1096.04	1.12	12152558.94
min	1.00	1.00	1.00	956703932.00
25%	1506.00	1030.00	3.00	965302637.00
50%	3070.00	1835.00	4.00	973018006.00
75%	4476.00	2770.00	4.00	975220939.00
max	6040.00	3952.00	5.00	1046454590.00

From the output of describe function and range of columns printed we can see that average rating is around 3.58 while median is around 4 so little deviation but not much.

Also, by inspecting the range of other columns we can say that there no erronous values present in our data.

[13]:
```
i = 1

for col in users.columns:
    if users[col].dtype == 'object':
        print(str(i) + '. "' + col + '" is categorical column with total unique␣
 ↪values as: ' + str(len(users[col].unique())))
```

```
    else:
        print(str(i) + '. "' + col + '" is continuous column with min, max and␣
↪mean values as: ', users[col].min(), users[col].max(), users[col].mean())
    i += 1
```

```
1. "UserID" is continuous column with min, max and mean values as:  1 6040
3020.5
2. "Gender" is categorical column with total unique values as: 2
3. "Age" is continuous column with min, max and mean values as:  1 56
30.639238410596025
4. "Occupation" is continuous column with min, max and mean values as:  0 20
8.146854304635761
5. "Zip-code" is categorical column with total unique values as: 3439
```

[14]:
```
users.describe()
```

[14]:

	UserID	Age	Occupation
count	6040.00	6040.00	6040.00
mean	3020.50	30.64	8.15
std	1743.74	12.90	6.33
min	1.00	1.00	0.00
25%	1510.75	25.00	3.00
50%	3020.50	25.00	7.00
75%	4530.25	35.00	14.00
max	6040.00	56.00	20.00

From the output of describe function and range of columns printed we can see that there are only 2 unique values in Gender column which indicates Male/Female and for column Zip-Code we can see that users are from diversified location.

Additionally, we can see in age we have minimum value as 1 but it is due to encoding we have done it actually indicates 18+ years.

The same is true for Occupation as actual values doesn't mean anything because this variable is also encoded.

[15]:
```
i = 1

for col in movies.columns:
    if movies[col].dtype == 'object':
        print(str(i) + '. "' + col + '" is categorical column with total unique␣
↪values as: ' + str(len(movies[col].unique())))
    else:
        print(str(i) + '. "' + col + '" is continuous column with min, max and␣
↪mean values as: ', movies[col].min(), movies[col].max(), movies[col].mean())
    i += 1
```

```
1. "Movie ID" is continuous column with min, max and mean values as:  1 3952
1986.0494463044038
```

2. "Title" is categorical column with total unique values as: 3883
3. "Genres" is categorical column with total unique values as: 301

[16]:
```
movies.describe()
```

[16]:
```
        Movie ID
count   3883.00
mean    1986.05
std     1146.78
min        1.00
25%      982.50
50%     2010.00
75%     2980.50
max     3952.00
```

From the output of describe function and range of columns we can see there are 301 types of genres which is sort of intriguing because in data genres are combined with | operator so their combinations are in abundance but in reality genres are very less around 19

Since, all data is scattered it would be better to bring everything on same level for better understanding and analysis

Feature Engineering

[17]:
```
ratings.isna().sum() * 100 / ratings.shape[0]
```

[17]:
```
UserID      0.00
MovieID     0.00
Rating      0.00
Timestamp   0.00
dtype: float64
```

[18]:
```
users.isna().sum() * 100 / users.shape[0]
```

[18]:
```
UserID       0.00
Gender       0.00
Age          0.00
Occupation   0.00
Zip-code     0.00
dtype: float64
```

[19]:
```
movies.isna().sum() * 100 / movies.shape[0]
```

[19]:
```
Movie ID   0.00
Title      0.00
Genres     0.00
dtype: float64
```

We can see that there are no null values present in our dataset. Hence, any merging operation is valid for us.

0.0.3 2. Data Pre-processing

[20]:
```
ratings['hour'] = ratings['Timestamp'].apply(lambda x: datetime.
 ↪fromtimestamp(x).hour)
ratings.head()
```

[20]:

	UserID	MovieID	Rating	Timestamp	hour
0	1	1193	5	978300760	3
1	1	661	3	978302109	4
2	1	914	3	978301968	4
3	1	3408	4	978300275	3
4	1	2355	5	978824291	5

Since, we have been provided with Timestamp in data so we converted to datetime using datetime and extracted hour to get information at what time mostly people are viewing content on the Zee website.

[21]:
```
users = users.merge(ratings.groupby('UserID').Rating.mean().reset_index(), on =␣
 ↪'UserID')
users = users.merge(ratings.groupby('UserID').hour.median().reset_index(), on =␣
 ↪'UserID')
users = users.merge(ratings.groupby('UserID').Rating.count().reset_index(), on␣
 ↪= 'UserID')
users = users.rename(columns = {"Rating_x" : "Rating", "Rating_y" :␣
 ↪"RatingCount"})
users.head()
```

[21]:

	UserID	Gender	Age	Occupation	Zip-code	Rating	hour	RatingCount
0	1	F	1	10	48067	4.19	4.00	53
1	2	M	56	16	70072	3.71	3.00	129
2	3	M	25	15	55117	3.90	2.00	51
3	4	M	45	7	02460	4.19	1.00	21
4	5	M	25	20	55455	3.15	12.00	198

Since, we have verified that there are no null values hence it is safer to merge columns. For merging, we found average Rating given by each user and then merged with respective User entry.

The same process we did for Hour column but it is better to take median to get integer and remove outliers (say a user views mostly at 1 AM but once he viewed at 11 PM so taking mean would skew but median would still give accurate results)

[22]:
```
moviesExploded = movies.copy()
moviesExploded['Genres'] = moviesExploded['Genres'].str.split('|')
moviesExploded = moviesExploded.explode('Genres')
moviesExploded.head()
```

[22]:

	Movie ID	Title	Genres
0	1	Toy Story (1995)	Animation
0	1	Toy Story (1995)	Children's
0	1	Toy Story (1995)	Comedy
1	2	Jumanji (1995)	Adventure
1	2	Jumanji (1995)	Children's

Since, in our movies dataframe the Genres column has combined Genres so for better analysis it was better to split into different rows i.e. converting data into 1NF form.

[23]:
```
merged = pd.merge(ratings, users, left_on = 'UserID', right_on = 'UserID', how␣
 ↪= "outer")
merged = pd.merge(merged, movies, left_on = 'MovieID', right_on = "Movie ID",␣
 ↪how = "inner").sort_values('UserID')
merged = merged[["UserID", "MovieID", "Timestamp", "Gender", "Age",␣
 ↪"Occupation",
                 "Zip-code", "Rating_y", "RatingCount", "hour_y", "Title",␣
 ↪"Genres"]]
merged = merged.rename(columns = {"Rating_y" : "Rating", "hour_y" : "hour"})
merged.reset_index(drop = True, inplace = True)
merged.head()
```

[23]:

	UserID	MovieID	Timestamp	Gender	Age	Occupation	Zip-code	Rating	\
0	1	1193	978300760	F	1	10	48067	4.19	
1	1	48	978824351	F	1	10	48067	4.19	
2	1	938	978301752	F	1	10	48067	4.19	
3	1	1207	978300719	F	1	10	48067	4.19	
4	1	1721	978300055	F	1	10	48067	4.19	

	RatingCount	hour	Title	\
0	53	4.00	One Flew Over the Cuckoo's Nest (1975)	
1	53	4.00	Pocahontas (1995)	
2	53	4.00	Gigi (1958)	
3	53	4.00	To Kill a Mockingbird (1962)	
4	53	4.00	Titanic (1997)	

	Genres
0	Drama
1	Animation|Children's|Musical|Romance
2	Musical
3	Drama
4	Drama|Romance

As, we have no null values so it would be better to combine all data into single one. For this process, initially we combined ratings and users by taking an outer join so that every entry is mapped and we named it as merged dataframe

Post that we combined merged and movies dataframe as inner join. Also, sorting of UserID is done

to get data in chronological order of Users.

Then we removed duplicate columns like Rating and hour as they appeared in both ratings and users and we kept values of users one only as we combined values based on user by taking mean and median.

Post that, we renamed columns due to merging to proper convention i.e. Rating_y has been changed to Rating and hour_y has been changed to hour

[24]:
```
merged.shape
```

[24]:
```
(1000209, 12)
```

Since, we merged 3 dataframe so the size of our final data is 1000209 rows and 12 columns which is pretty huge.

[25]:
```
merged.describe()
```

[25]:

	UserID	MovieID	Timestamp	Age	Occupation	Rating	\
count	1000209.00	1000209.00	1000209.00	1000209.00	1000209.00	1000209.00	
mean	3024.51	1865.54	972243695.40	29.74	8.04	3.58	
std	1728.41	1096.04	12152558.94	11.75	6.53	0.44	
min	1.00	1.00	956703932.00	1.00	0.00	1.02	
25%	1506.00	1030.00	965302637.00	25.00	2.00	3.32	
50%	3070.00	1835.00	973018006.00	25.00	7.00	3.61	
75%	4476.00	2770.00	975220939.00	35.00	14.00	3.88	
max	6040.00	3952.00	1046454590.00	56.00	20.00	4.96	

	RatingCount	hour
count	1000209.00	1000209.00
mean	389.91	9.50
std	324.74	6.75
min	20.00	0.00
25%	147.00	4.00
50%	302.00	8.00
75%	544.00	13.00
max	2314.00	23.00

From above describe function we can see that columns are having different scales as columns like RatingCount is in thousands while hour, Rating is on scale of tens

[26]:
```
merged.isna().sum() * 100 / merged.shape[0]
```

[26]:
```
UserID        0.00
MovieID       0.00
Timestamp     0.00
Gender        0.00
Age           0.00
Occupation    0.00
```

```
Zip-code      0.00
Rating        0.00
RatingCount   0.00
hour          0.00
Title         0.00
Genres        0.00
dtype: float64
```

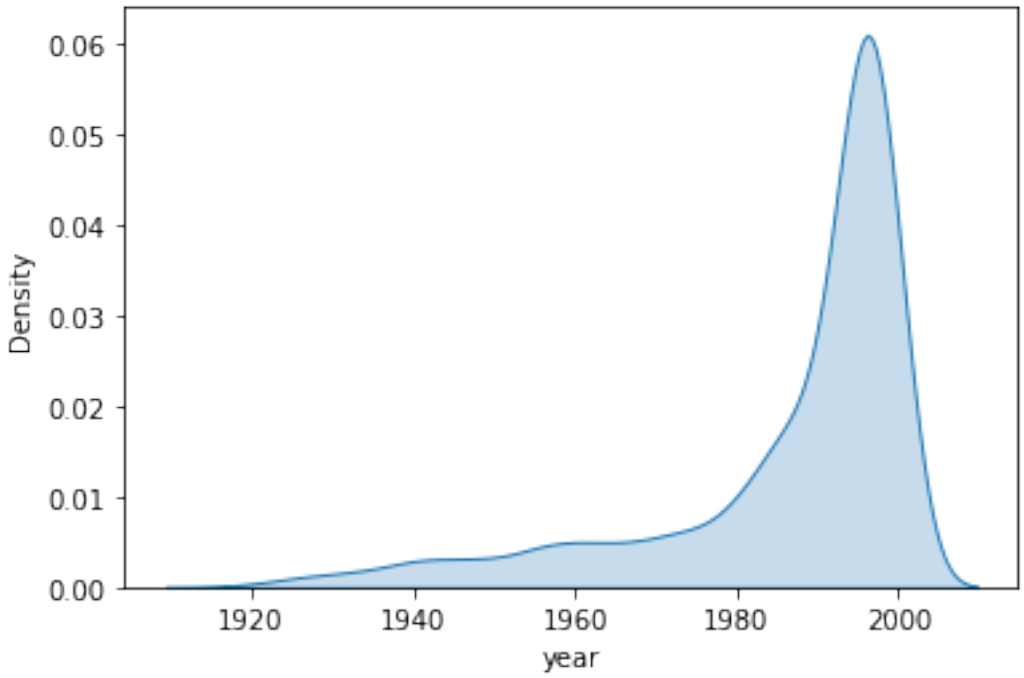
We can see that there are no null values present in our data.

0.0.4 3. Univariate Analysis

[27]:
```
movies['year'] = movies['Title'].apply(lambda x : x.split('(')[-1].replace(')',␣
↪''))
movies['year'] = movies['year'].astype(int)
```

We split the year column to extract in which year the movie was released.

[28]:
```
sns.kdeplot(movies['year'], fill = True)
plt.show()
```



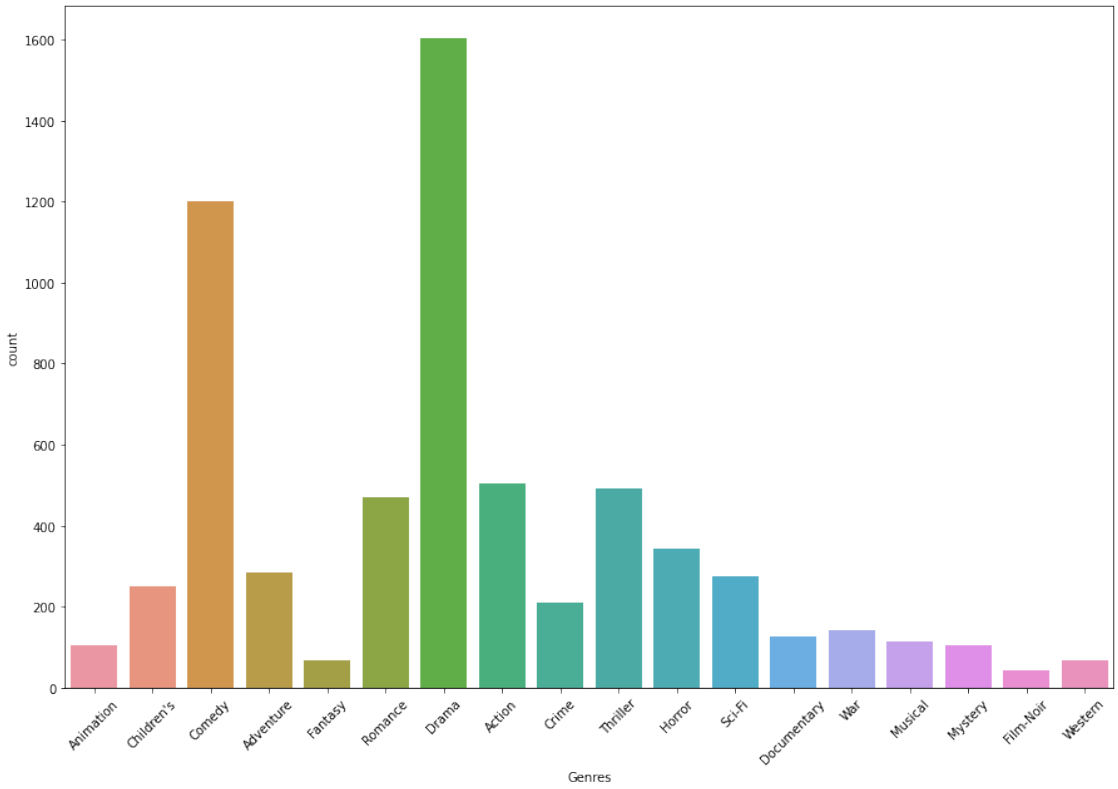
Observation

From above countplot we can see that most of the movies are from year 1990 - 2000

Hence, most of the movies which are present in our dataset are from 90s and the above countplot

proves that

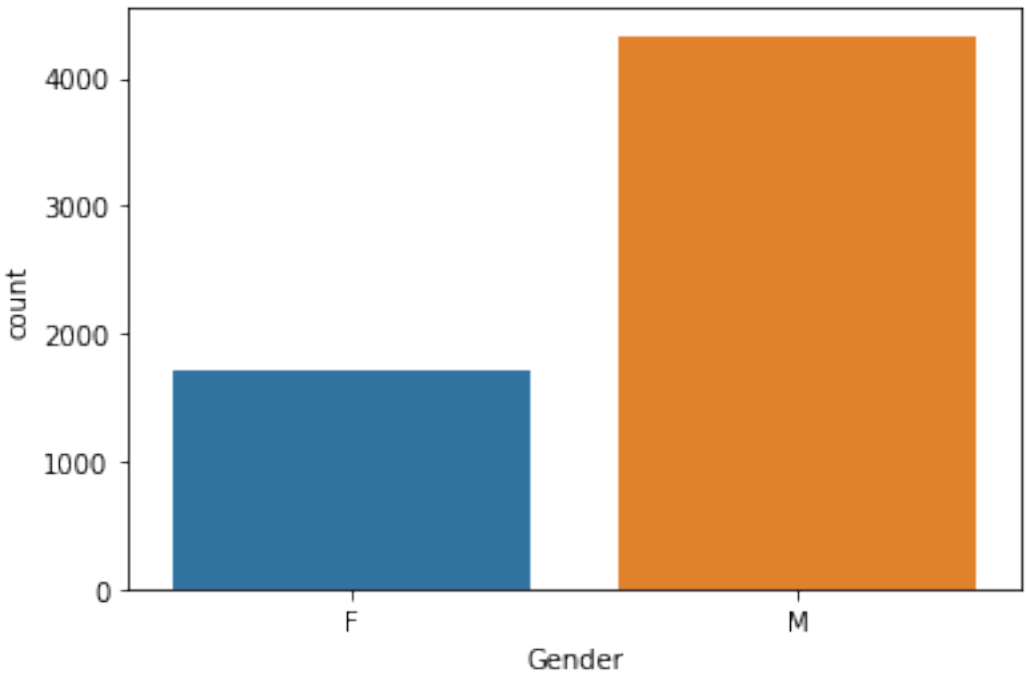
[29]:
```
fig = plt.figure(figsize = (15, 10))
sns.countplot(moviesExploded['Genres'])
plt.xticks(rotation = 45)
plt.show()
```



Observation

From above countplot we can say that most of the movies are of genre Drama genre and second top genre is Comedy

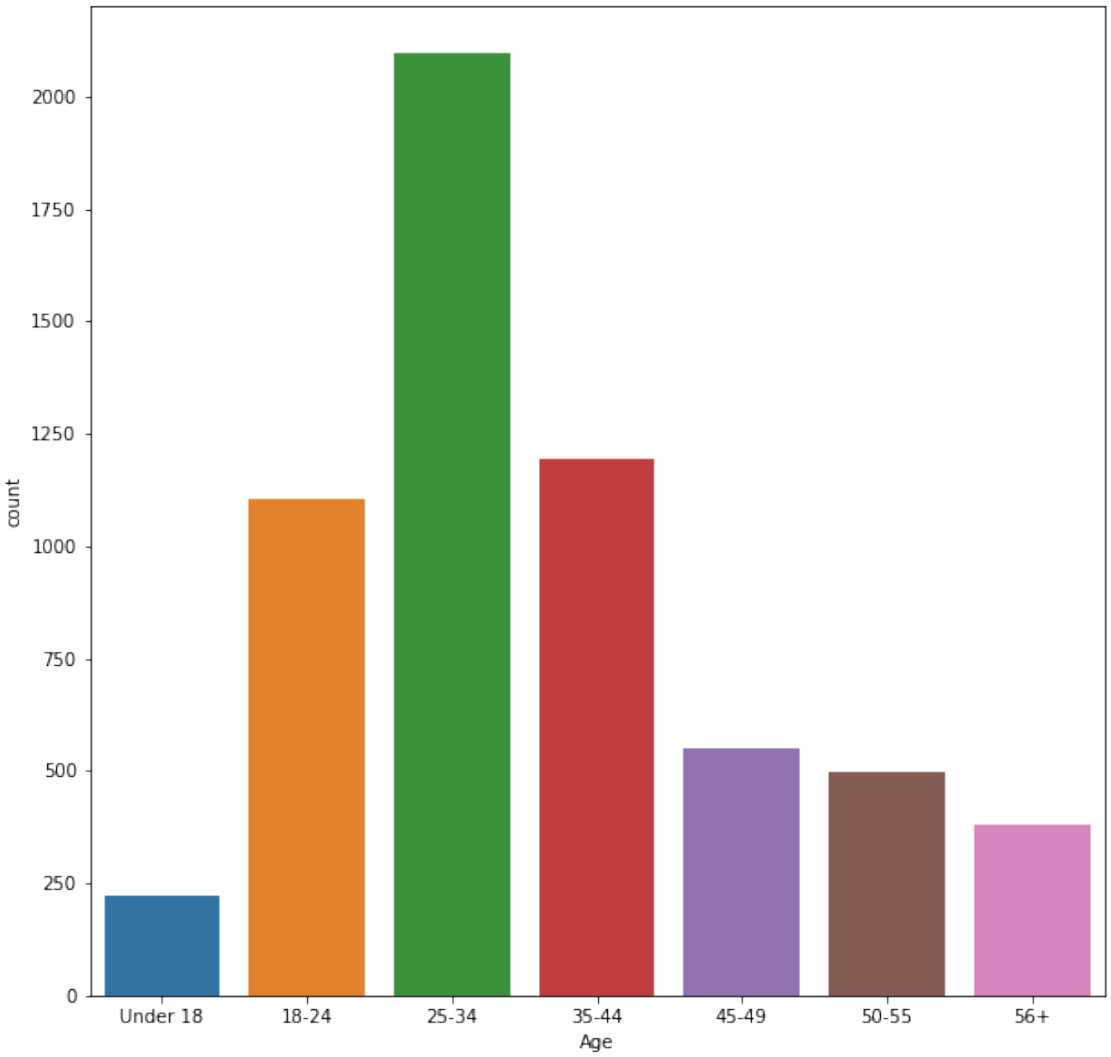
[30]:
```
sns.countplot(users['Gender'])
plt.show()
```



Observation

From above countplot we can see that most of the users in our database are Males which are way higher than Female viewers

[31]:
```
fig = plt.figure(figsize = (10, 10))
sns.countplot(users['Age'])
plt.xticks([0, 1, 2, 3, 4, 5, 6],
           ['Under 18', '18-24', '25-34', '35-44', '45-49', '50-55', '56+'])
plt.show()
```

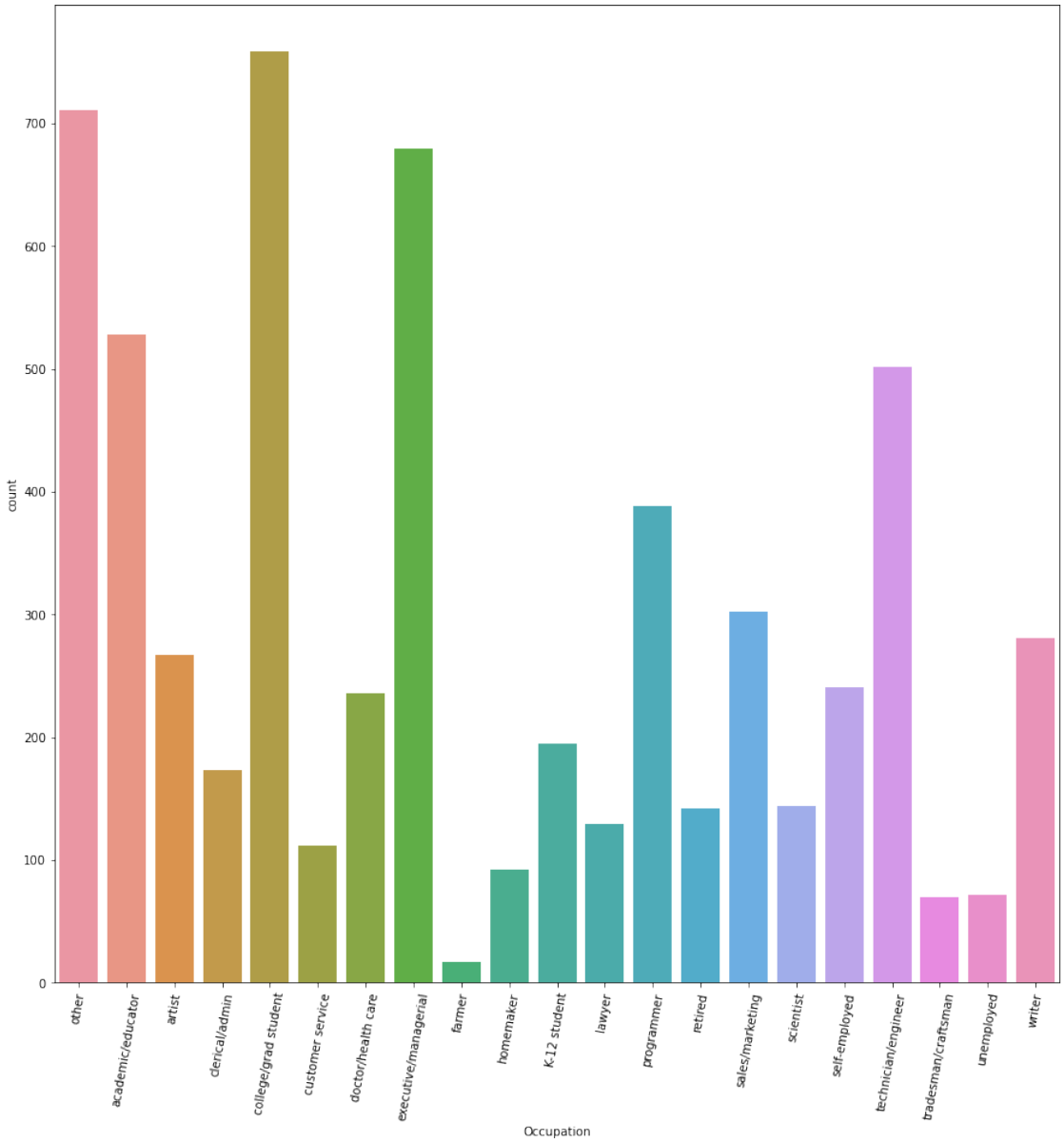


Observation

From above countplot we can see that most of the users present in our database are of age group 25-34 years old and which makes sense because most of the youth are attracted towards OTT platform while Under 18 are less in number due to parents restrictions or focus on studies.

[32]:
```
fig = plt.figure(figsize = (15, 15))
sns.countplot(users['Occupation'])
plt.xticks(np.arange(21),
           ['other', 'academic/educator', 'artist', 'clerical/admin', 'college/
 ↪grad student',
            'customer service', 'doctor/health care', 'executive/managerial',␣
 ↪'farmer', 'homemaker',
            'K-12 student', 'lawyer', 'programmer', 'retired', 'sales/
 ↪marketing', 'scientist',
```

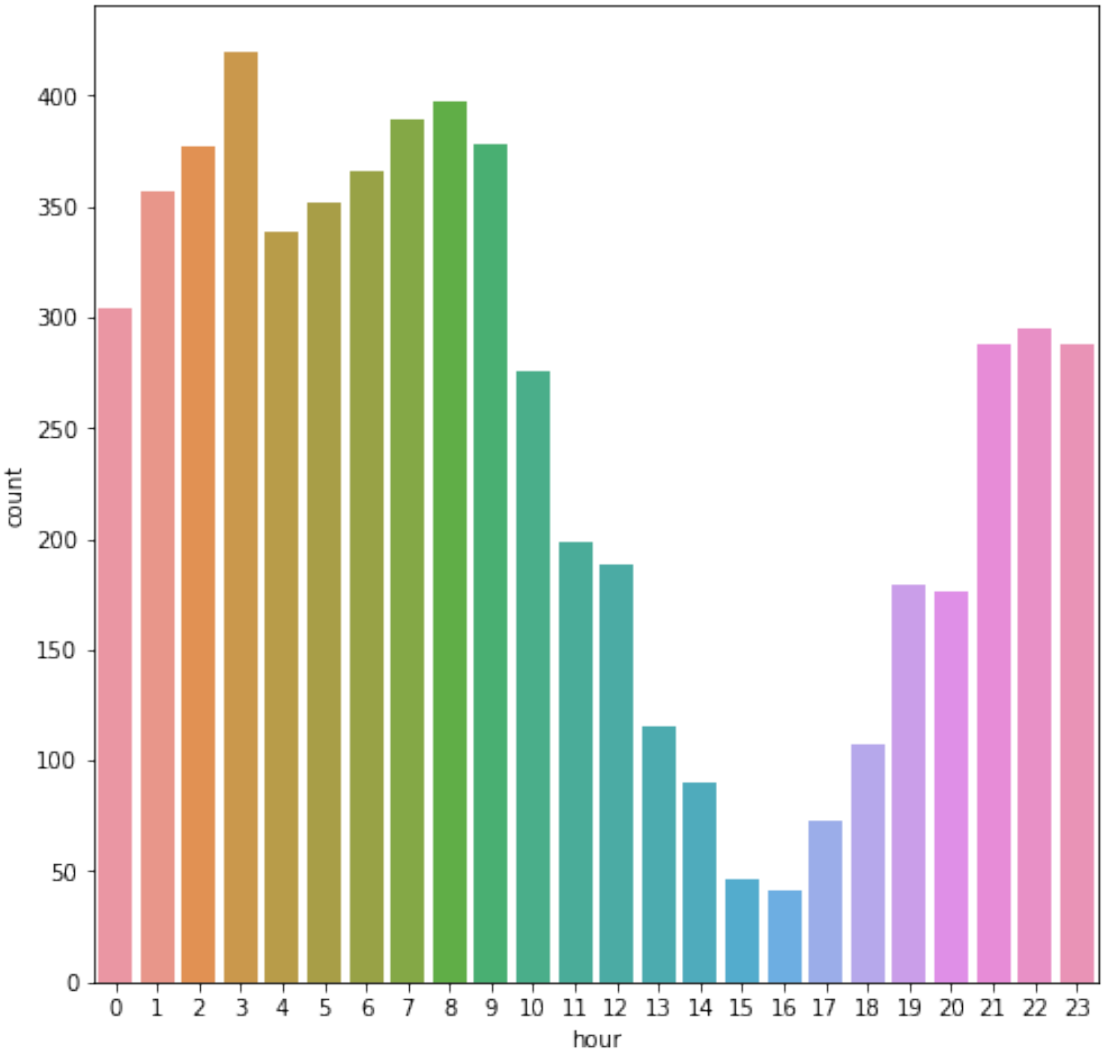
```
            'self-employed', 'technician/engineer', 'tradesman/craftsman',␣
 ↪'unemployed', 'writer'],
           rotation = 80)
plt.show()
```



Observation

From above countplot we can see that most of the users in our system are college/grad student and it makes sense because they are not working so have work load and have little time so they prefer OTT platforms and also most of them may be in hostel or PGs so to pass their time they look forward to OTT platforms.

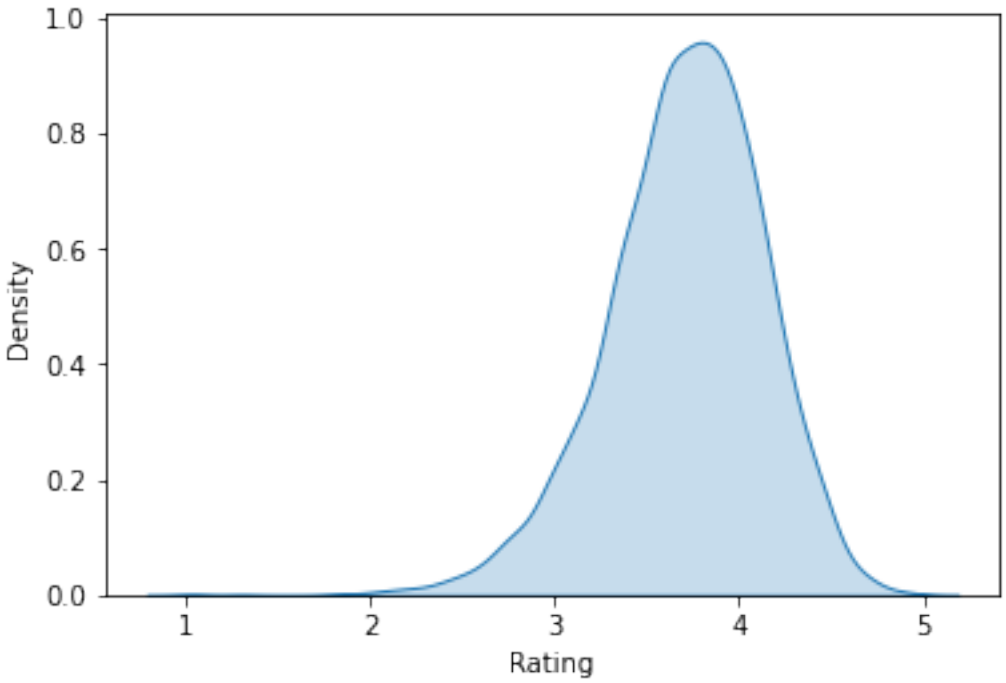
[33]:
```
fig = plt.figure(figsize = (8, 8))
sns.countplot(users['hour'].astype(int))
plt.show()
```



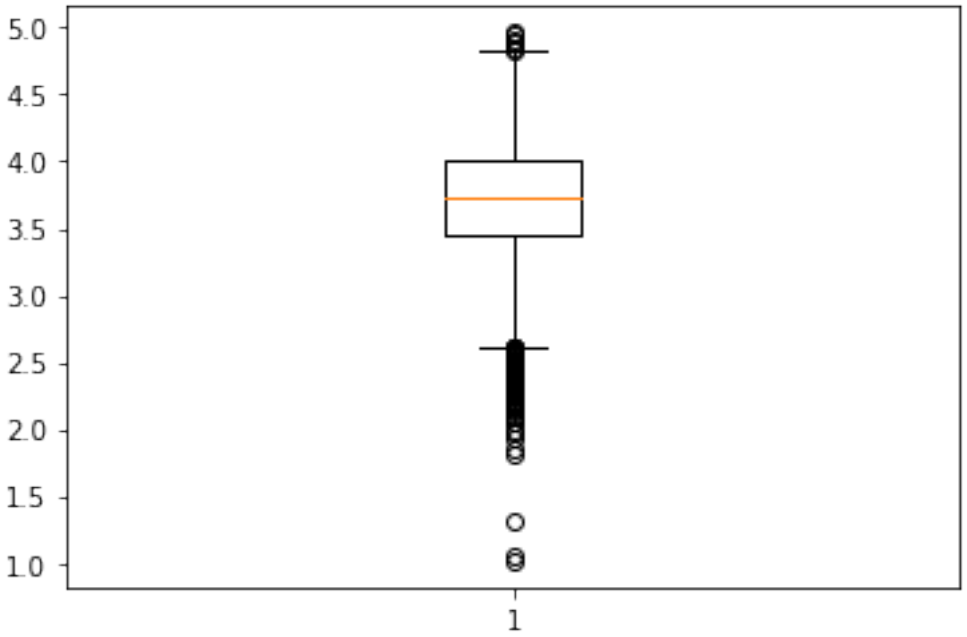
Observation

We can see that most of the people are viewing our content at 3 AM in morning and in afternoon the content view is very less which indicates that mostly people are busy be it college grad or working professionals in afternoon hence they either watch early morning or late nights.

[34]:
```
sns.kdeplot(users['Rating'], fill = True)
plt.show()
```



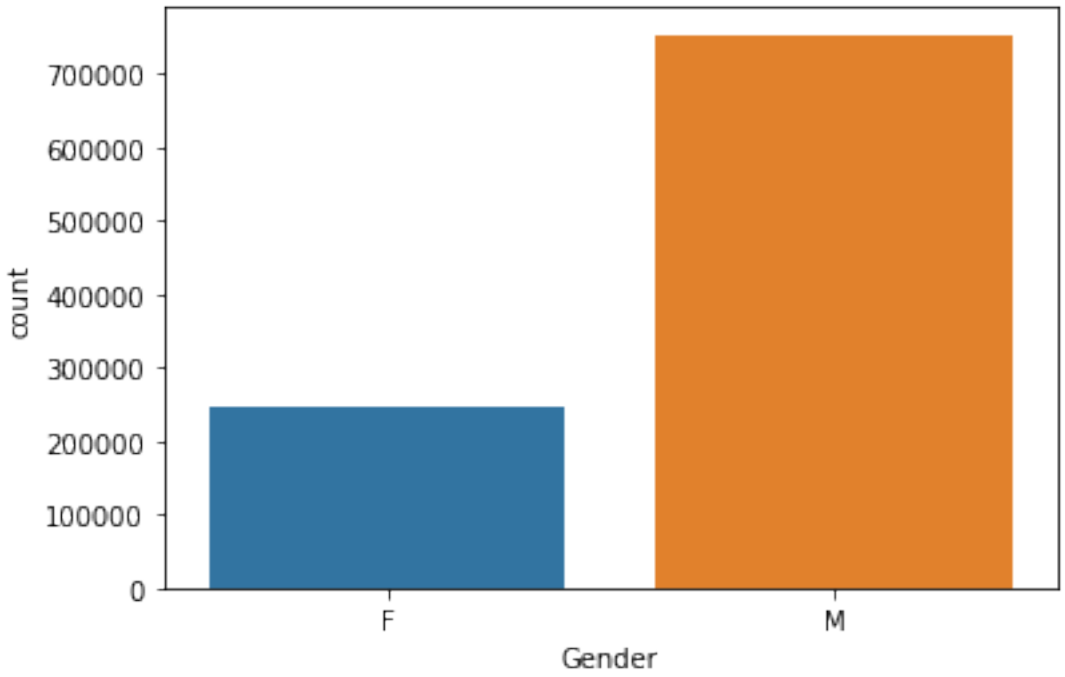
[35]:
```
plt.boxplot(users['Rating'])
plt.show()
```



Observation

We can see that there are several outliers present in Rating column as most people have given rating near to 4 but it would be unwise to remove them since we have to make recommendation system and for that user preference is very important as if someone disliked the movie then similar movies to it won't be recommended to him.

[36]:
```
sns.countplot(merged['Gender'])
plt.show()
```



[37]:
```
merged.groupby('Gender').mean().Rating
```

[37]:
```
Gender
F    3.62
M    3.57
Name: Rating, dtype: float64
```

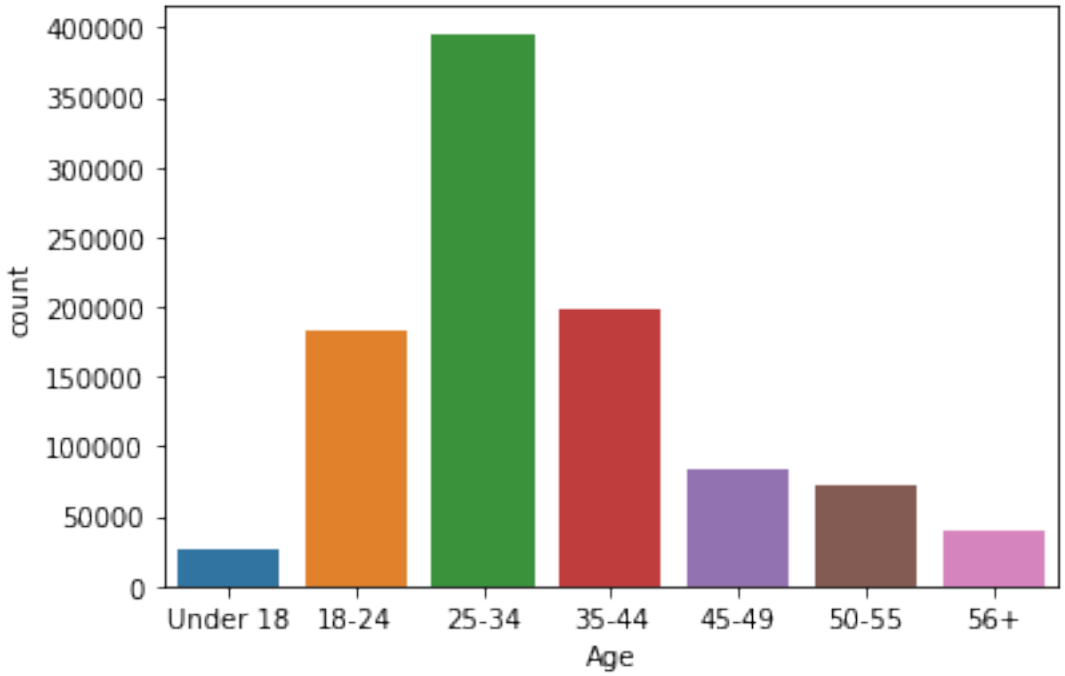
Observation

Since, in our merged dataframe we have different MovieID for UserID and minimum rating is around 1 hence on finding countplot we can find out which gender has watched most movies and rated them.

Hence, males have watched more movies and rated them as compared to Females

But females have like movies more than Males as the mean rating given by females are 1.4% more than that of Males.

[38]:
```
sns.countplot(merged['Age'])
plt.xticks([0, 1, 2, 3, 4, 5, 6],
           ['Under 18', '18-24', '25-34', '35-44', '45-49', '50-55', '56+'])
plt.show()
```



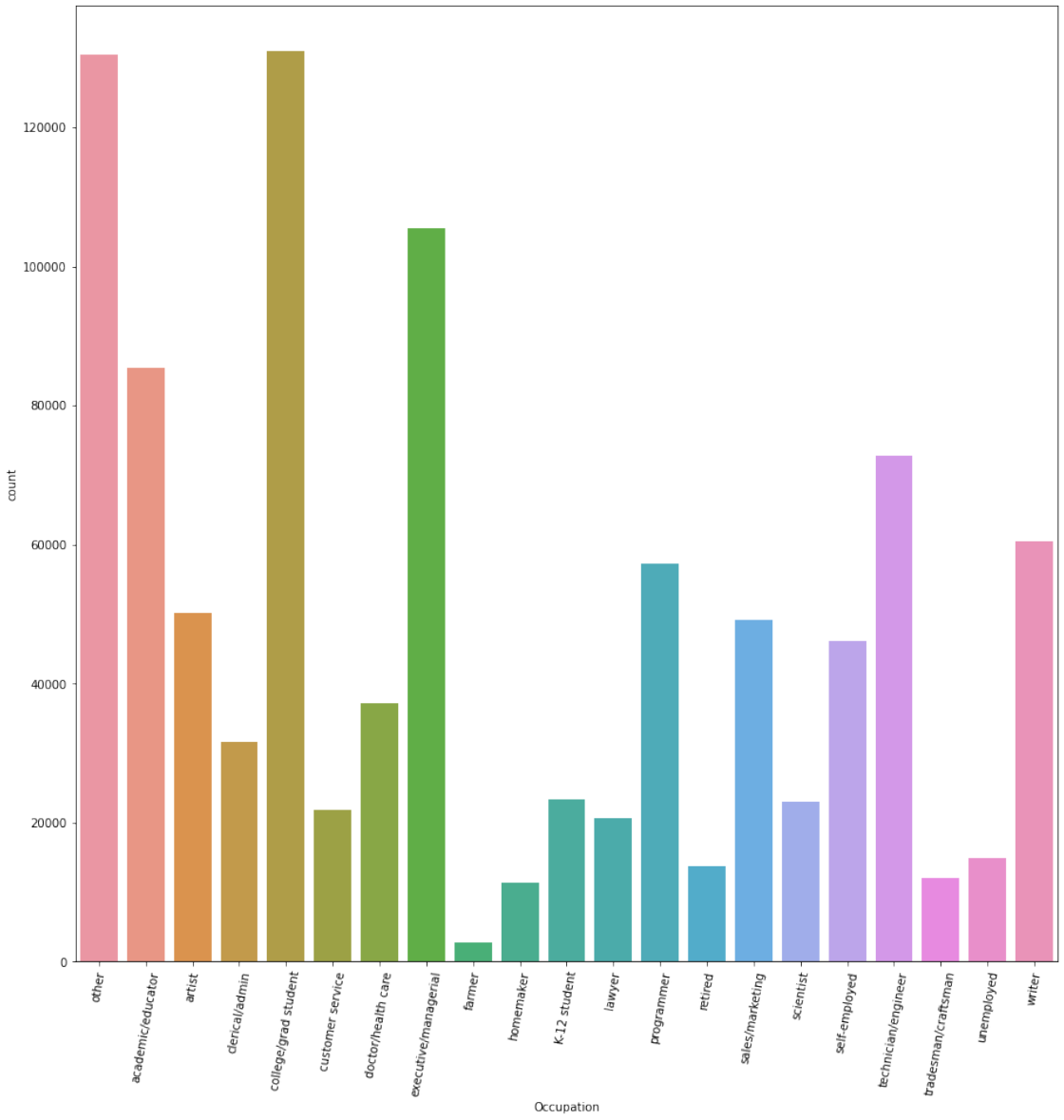
Observation

Since, in our merged dataframe we have different MovieID for UserID and minimum rating is around 1 hence for countplot we can find out which age group has watched most movies and rated them.

So, people of age group 25-34 have watched more movies and rated them as compared to other peers and it make sense because from above plots these users are in abundance so they must have given more ratings then other

[39]:
```
fig = plt.figure(figsize = (15, 15))
sns.countplot(merged['Occupation'])
plt.xticks(np.arange(21),
           ['other', 'academic/educator', 'artist', 'clerical/admin', 'college/
 ↪grad student',
            'customer service', 'doctor/health care', 'executive/managerial',␣
 ↪'farmer', 'homemaker',
            'K-12 student', 'lawyer', 'programmer', 'retired', 'sales/
 ↪marketing', 'scientist',
            'self-employed', 'technician/engineer', 'tradesman/craftsman',␣
 ↪'unemployed', 'writer'],
```

```
            rotation = 80)
plt.show()
```



Observation

Since, in our merged dataframe we have different MovieID for UserID and minimum rating is around 1 hence for countplot we can find out which profession (people belonging to differnt professions) has watched most movies and rated them.

So, people of profession college/grad student and other have watched more movies and rated them as compared to other peers

0.0.5 4. Bivariate Analysis

[40]:
```
mergedExploded = merged.copy()
mergedExploded['Genres'] = mergedExploded['Genres'].str.split('|')
mergedExploded = mergedExploded.explode('Genres')
mergedExploded.head()
```

[40]:

	UserID	MovieID	Timestamp	Gender	Age	Occupation	Zip-code	Rating	\
0	1	1193	978300760	F	1	10	48067	4.19	
1	1	48	978824351	F	1	10	48067	4.19	
1	1	48	978824351	F	1	10	48067	4.19	
1	1	48	978824351	F	1	10	48067	4.19	
1	1	48	978824351	F	1	10	48067	4.19	

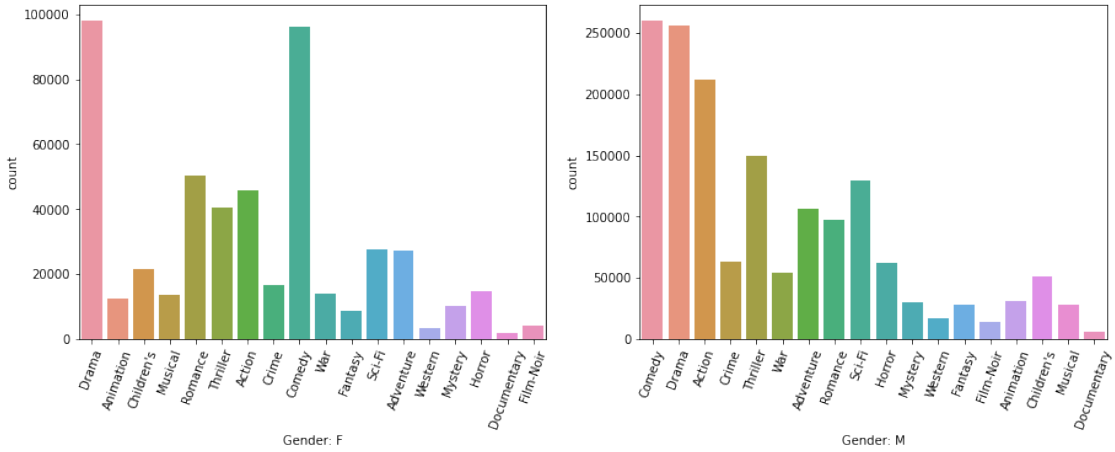
	RatingCount	hour	Title	Genres
0	53	4.00	One Flew Over the Cuckoo's Nest (1975)	Drama
1	53	4.00	Pocahontas (1995)	Animation
1	53	4.00	Pocahontas (1995)	Children's
1	53	4.00	Pocahontas (1995)	Musical
1	53	4.00	Pocahontas (1995)	Romance

Since, to make analysis about genres it is important to split genres since movies have genres combined with | symbol. This would help to make better analysis and understanding of user.

[41]:
```
counter = 1
fig = plt.figure(figsize = (15, 5))

for g in mergedExploded['Gender'].unique():
    plt.subplot(1, 2, counter)
    sns.countplot(mergedExploded[mergedExploded['Gender'] == g]['Genres'])
    plt.xticks(rotation = 70)
    counter += 1
    plt.xlabel(f'Gender: {g}')

plt.show()
```



Recommendation

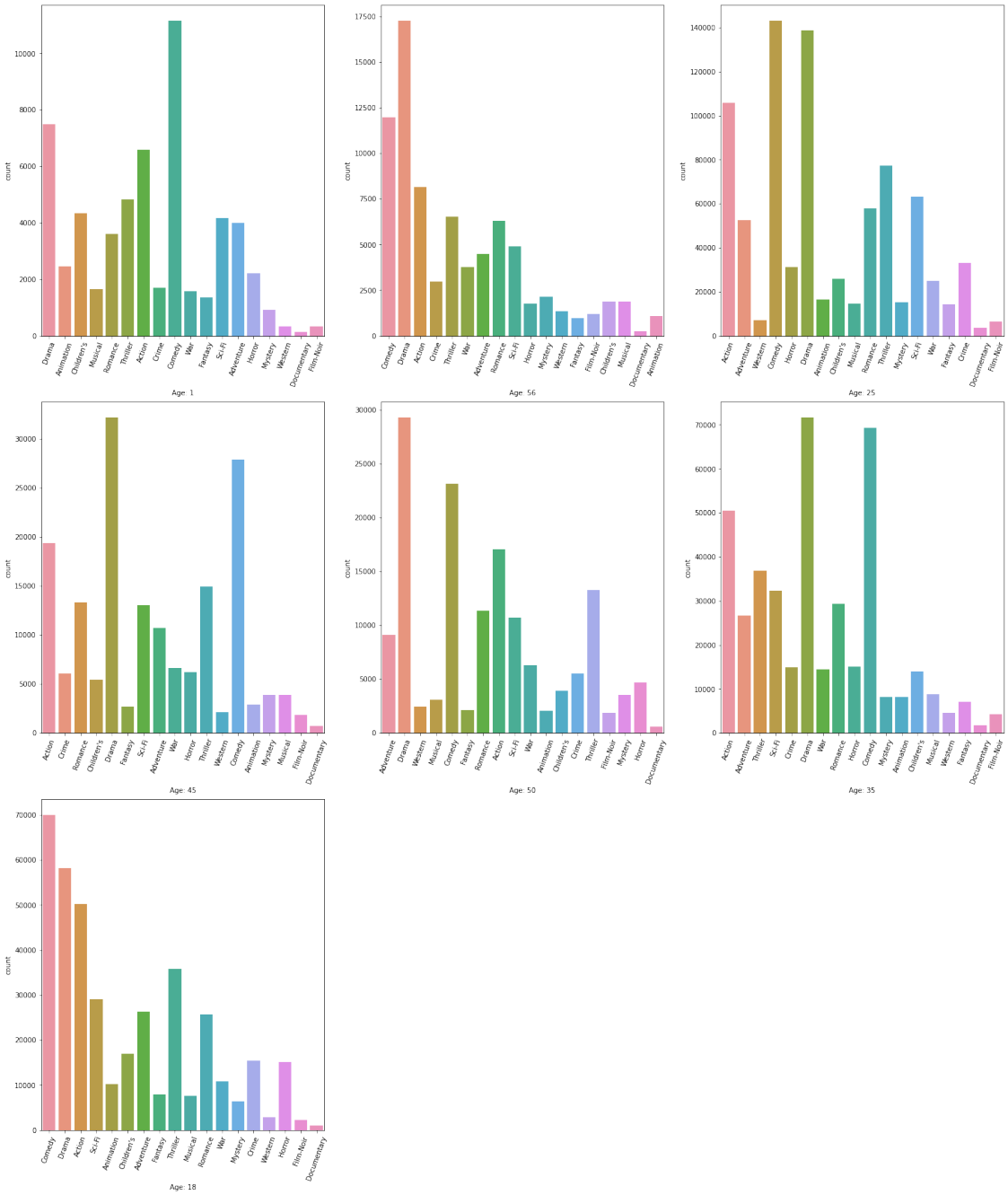
From above two countplots we can provide recommendations gender wise to each user

- For Females
  - Comedy movies
  - Drama movies
  - Romance movies
- For Males
  - Comedy movies
  - Drama movies
  - Action movies

[42]:
```
counter = 1
fig = plt.figure(figsize = (25, 30))

for age in mergedExploded['Age'].unique():
    plt.subplot(3, 3, counter)
    sns.countplot(mergedExploded[mergedExploded['Age'] == age]['Genres'])
    plt.xticks(rotation = 70)
    counter += 1
    plt.xlabel(f'Age: {age}')

plt.show()
```



Recommendation

From above 7 countplots plotted for different Age groups below are top 4 recommendations for each age bracket

- For age 1 (i.e. Under 18 years old)
  - Comedy
  - Drama

    - Action
    - Thriller
- For age 18 (18-24 years old)
    - Comedy
    - Drama
    - Action
    - Sci - Fi
- For age 25 (25-34 years old)
    - Comedy
    - Drama
    - Action
    - Thriller
- For age 35 (35-44 years old)
    - Comedy
    - Drama
    - Action
    - Thriller
- For age 45 (45-49 years old)
    - Comedy
    - Drama
    - Action
    - Thriller
- For age 50 (50-55 years old)
    - Comedy
    - Drama
    - Action
    - Thriller
- For age 56 (56+ years old)
    - Comedy
    - Drama
    - Action
    - Romance

[43]:
```
topkratedmovie = 1

pd.DataFrame(merged.groupby('Title').mean().Rating.nlargest(topkratedmovie))
```

[43]:
```
                        Rating
Title
Condition Red (1995)      4.39
```

The movie Condition Red is the top rated movie among all movies and it has a rating of 4.39

[44]:
```
pd.DataFrame(mergedExploded.groupby('Genres').mean().Rating).
↪sort_values('Rating', ascending = False)
```

[44]:
```
              Rating
Genres
```

```
Film-Noir      3.68
Documentary    3.65
War            3.64
Drama          3.62
Musical        3.61
Mystery        3.60
Western        3.60
Crime          3.60
Romance        3.60
Animation      3.59
Fantasy        3.58
Adventure      3.57
Thriller       3.57
Sci-Fi         3.57
Comedy         3.57
Action         3.57
Children's     3.55
Horror         3.50
```

Observation

From above data we can see that Film-Noir is most rated genre while Horror is genre with less rating.

Recommendation

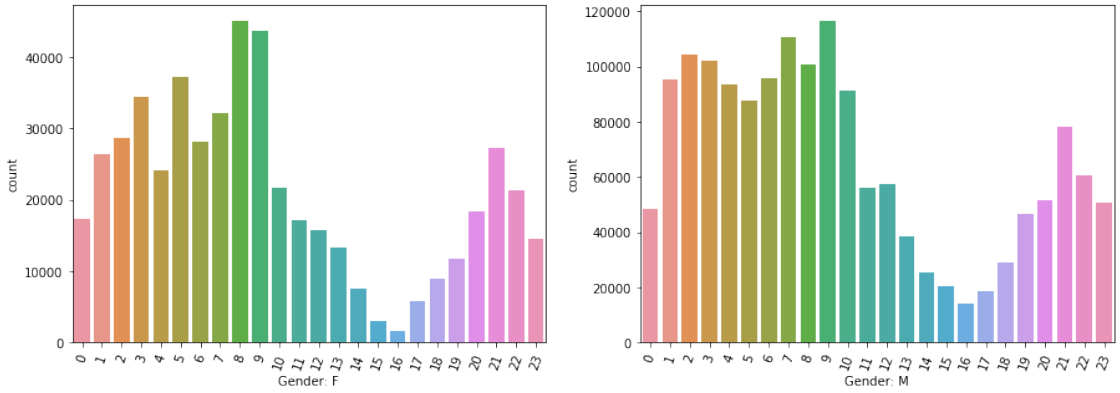
Hence, to make more user engagement below genres can be recommended as it has high average rating

- Film-Noir
- Documentary
- War

[45]:
```
counter = 1
fig = plt.figure(figsize = (15, 5))

for g in mergedExploded['Gender'].unique():
    plt.subplot(1, 2, counter)
    sns.countplot(mergedExploded[mergedExploded['Gender'] == g]['hour'].
 ↪astype(int))
    plt.xticks(rotation = 70)
    counter += 1
    plt.xlabel(f'Gender: {g}')

plt.show()
```



Observation

From above countplots we can see that people from both genders i.e. M and F are almost active at almost same time intervals.

0.0.6 5. Content Based Recommendation System (using Pearson Correlation)

5.1 Item - Item based similarity and recommendation Since, we need to find attributes of features hence we will use pivot table attribute to generate dataframe where each movie has features as 0 or 1 against each genre.

So, if a movie has 1 against genre Animation it means the genre of movie is animation. This will help us by making an algorithm that will find simialrity between items.

Also, we can also add year but I am not using since if a movie launched in year 1980 say Top Gun and same sequel of movie i.e. Top Gun : Maverick . So by adding year these two movies may be distant apart but second one is sequel of first so it has same genres and to get it into top recommendation I choose not to include year

[46]:
```
moviesExplodedpivotted = moviesExploded.pivot(index = 'Movie ID', columns =␣
↪'Genres', values = 'Title')
moviesExplodedpivotted = ~moviesExplodedpivotted.isna()
moviesExplodedpivotted = moviesExplodedpivotted.astype(int)
moviesExplodedpivotted.head()
```

[46]:
```
Genres    Action  Adventure  Animation  Children's  Comedy  Crime  \
Movie ID
1              0          0          1           1       1      0
2              0          1          0           1       0      0
3              0          0          0           0       1      0
4              0          0          0           0       1      0
5              0          0          0           0       1      0

Genres    Documentary  Drama  Fantasy  Film-Noir  Horror  Musical  Mystery  \
Movie ID
```

```
1                   0      0         0          0       0         0         0
2                   0      0         1          0       0         0         0
3                   0      0         0          0       0         0         0
4                   0      1         0          0       0         0         0
5                   0      0         0          0       0         0         0

Genres    Romance  Sci-Fi  Thriller  War  Western
Movie ID
1               0       0         0    0        0
2               0       0         0    0        0
3               1       0         0    0        0
4               0       0         0    0        0
5               0       0         0    0        0
```

[47]:
```
def findPearsonCorrelationItemSimilarity(movieName, noofrecommendations = 5):
    _movies = movies[movies['Title'].str.contains(movieName)]

    if _movies.shape[0] == 0:
        return 'No movie with given name present'

    distances = []
    _movieIndex = _movies['Movie ID'].iloc[0]
    for movie in moviesExplodedpivotted.index:
        if _movieIndex == movie:
            continue
        _pearsonCorr, _ = pearsonr(moviesExplodedpivotted.loc[_movieIndex],␣
↪moviesExplodedpivotted.loc[movie])
        distances.append([_movieIndex, movie, _pearsonCorr])

    movieRanked = pd.DataFrame(distances, columns=['QueryMovie', 'OtherMovies',␣
↪'PearsonDistance'])
    movieRanked = movieRanked.merge(movies[['Movie ID', 'Title']], left_on =␣
↪'QueryMovie', right_on = 'Movie ID').rename(columns = {'Title':␣
↪'QueryMovieTitle'}).drop(columns = ['Movie ID'])
    movieRanked = movieRanked.merge(movies[['Movie ID', 'Title']], left_on =␣
↪'OtherMovies', right_on = 'Movie ID').rename(columns = {'Title':␣
↪'RecommendedMoviesTitle'}).drop(columns = ['Movie ID'])
    movieRanked = movieRanked.sort_values(by = 'PearsonDistance', ascending =␣
↪False)
    movieRanked = movieRanked.reset_index()
    movieRanked = movieRanked[['QueryMovieTitle', 'RecommendedMoviesTitle',␣
↪'PearsonDistance']]
    return movieRanked.head(noofrecommendations)
```

[48]:
```
findPearsonCorrelationItemSimilarity("Liar Liar")
```

[48]:

	QueryMovieTitle	RecommendedMoviesTitle	PearsonDistance
0	Liar Liar (1997)	Election (1999)	1.00
1	Liar Liar (1997)	Mystery, Alaska (1999)	1.00
2	Liar Liar (1997)	Sour Grapes (1998)	1.00
3	Liar Liar (1997)	Cops and Robbersons (1994)	1.00
4	Liar Liar (1997)	Dazed and Confused (1993)	1.00

The range of Pearson correlation is -1 to 1 where 1 indicates strong positive correlation and 0 means no correlation or similarity.

Hence, for movie Liar Liar the top 5 recommended movies are

- Election
- Mystery, Alaska
- Sour Grapes
- Cops and Robbersons
- Dazed and Confused

A function is defined above where one needs to pass a movie name and how many recommendation required (optional) and it will return top k similar movies to that movie

5.2 User - User based similarity and recommendation

[49]:
```
users['GenderMapped'] = users['Gender'].map({"F" : 1, "M" : 0})
users.head()
```

[49]:

	UserID	Gender	Age	Occupation	Zip-code	Rating	hour	RatingCount	\
0	1	F	1	10	48067	4.19	4.00	53	
1	2	M	56	16	70072	3.71	3.00	129	
2	3	M	25	15	55117	3.90	2.00	51	
3	4	M	45	7	02460	4.19	1.00	21	
4	5	M	25	20	55455	3.15	12.00	198	

	GenderMapped
0	1
1	0
2	0
3	0
4	0

For using above data on any algorithm the gender is converted to binary value and stored into new column called GenderMapped.

For our user - user recommendation system, we will use columns

- GenderMapped
- Age
- Occupation
- Rating
- hour

I am not taking Zip - code for now but we can take it for making demographics recommendation and also RatingCount since, some users watch lot of movies and some watch some so it would skew data sometimes.

Also, since scale of data is different it would be better to apply StandardScaler.

[50]:
```
userscaler = StandardScaler()

userfeatures = pd.DataFrame(userscaler.fit_transform(users[['Age',␣
 ↪'Occupation', 'GenderMapped',
                                                            'Rating', 'hour']]),
                            columns = ['Age', 'Occupation', 'GenderMapped',␣
 ↪'Rating', 'hour'],
                            index = users.set_index('UserID').index)
userfeatures.head()
```

[50]:

UserID	Age	Occupation	GenderMapped	Rating	hour
1	-2.30	0.29	1.59	1.13	-0.77
2	1.97	1.24	-0.63	0.02	-0.91
3	-0.44	1.08	-0.63	0.46	-1.05
4	1.11	-0.18	-0.63	1.14	-1.19
5	-0.44	1.87	-0.63	-1.29	0.35

[51]:
```
def findPearsonCorrelationUserSimilarity(userID, noofrecommendations = 5):
    distances = []
    for user in userfeatures.index:
        _pearsonCorr, _ = pearsonr(userfeatures.loc[userID], userfeatures.
 ↪loc[user])
        distances.append(_pearsonCorr)

    userRanked = pd.DataFrame()
    userRanked['SimilarUserID'] = users.set_index('UserID').index
    userRanked['PearsonDistance'] = distances
    userRanked = userRanked[userRanked['SimilarUserID'] != userID]
    userRanked = userRanked.sort_values(by = 'PearsonDistance', ascending =␣
 ↪False)
    userRanked.reset_index(inplace = True)
    userRanked = userRanked[["SimilarUserID", "PearsonDistance"]]
    return userRanked.head(noofrecommendations)
```

[52]:
```
findPearsonCorrelationUserSimilarity(50, noofrecommendations = 5)
```

[52]:

	SimilarUserID	PearsonDistance
0	1394	1.00
1	1076	0.99
2	2739	0.99
3	1813	0.99

```
4            1920              0.99
```

Hence, we built a user - user similar recommendation where a function is created which accepts userID and no of top similar users to be returned (optional) and it finds pearson correlation among all users and returns top k similar user to our target user.

I tried finding out similarity matrix between items and users but due to large computations my system was getting hanged. Hence, below i am attaching a smaller representation of how it will look like for both users and movies.

Hence, I am doing all calculations for top 20 movies and 20 users.

[53]:
```
distances = []
for movie1 in moviesExplodedpivotted.head(20).index:
    for movie2 in moviesExplodedpivotted.head(20).index:
        if movie1 == movie2:
            continue
        _pearsonCorr, _ = pearsonr(moviesExplodedpivotted.loc[movie1],␣
 ↪moviesExplodedpivotted.loc[movie2])
        distances.append([movie1, movie2, _pearsonCorr])
```

[54]:
```
itemsSimilarityMatrix = pd.DataFrame(distances, columns = ["Movie1", "Movie2",␣
 ↪"PearsonDistance"])
pd.pivot_table(values = 'PearsonDistance', index = "Movie1", columns =␣
 ↪"Movie2", data = itemsSimilarityMatrix).fillna(1)
```

[54]:
```
Movie2    1     2     3     4     5     6     7     8     9     10    11  \
Movie1
1       1.00  0.20  0.32  0.32  0.54 -0.20  0.32  0.32 -0.11 -0.20  0.20
2       0.20  1.00 -0.16 -0.16 -0.11 -0.20 -0.16  0.79 -0.11  0.20 -0.20
3       0.32 -0.16  1.00  0.44  0.69 -0.16  1.00 -0.12 -0.09 -0.16  0.79
4       0.32 -0.16  0.44  1.00  0.69 -0.16  0.44 -0.12 -0.09 -0.16  0.79
5       0.54 -0.11  0.69  0.69  1.00 -0.11  0.69 -0.09 -0.06 -0.11  0.54
6      -0.20 -0.20 -0.16 -0.16 -0.11  1.00 -0.16 -0.16  0.54  0.60 -0.20
7       0.32 -0.16  1.00  0.44  0.69 -0.16  1.00 -0.12 -0.09 -0.16  0.79
8       0.32  0.79 -0.12 -0.12 -0.09 -0.16 -0.12  1.00 -0.09  0.32 -0.16
9      -0.11 -0.11 -0.09 -0.09 -0.06  0.54 -0.09 -0.09  1.00  0.54 -0.11
10     -0.20  0.20 -0.16 -0.16 -0.11  0.60 -0.16  0.32  0.54  1.00 -0.20
11      0.20 -0.20  0.79  0.79  0.54 -0.20  0.79 -0.16 -0.11 -0.20  1.00
12      0.32 -0.16  0.44  0.44  0.69 -0.16  0.44 -0.12 -0.09 -0.16  0.32
13      0.79  0.32 -0.12 -0.12 -0.09 -0.16 -0.12  0.44 -0.09 -0.16 -0.16
14     -0.11 -0.11 -0.09  0.69 -0.06 -0.11 -0.09 -0.09 -0.06 -0.11  0.54
15     -0.20  0.20  0.32 -0.16 -0.11  0.20  0.32  0.32  0.54  0.60  0.20
16     -0.16 -0.16 -0.12  0.44 -0.09  0.32 -0.12 -0.12 -0.09  0.32  0.32
17     -0.16 -0.16  0.44  0.44 -0.09 -0.16  0.44 -0.12 -0.09 -0.16  0.79
18     -0.11 -0.11 -0.09 -0.09 -0.06  0.54 -0.09 -0.09 -0.06  0.54 -0.11
19      0.54 -0.11  0.69  0.69  1.00 -0.11  0.69 -0.09 -0.06 -0.11  0.54
20     -0.11 -0.11 -0.09 -0.09 -0.06  0.54 -0.09 -0.09  1.00  0.54 -0.11
```

```
Movie2    12    13    14    15    16    17    18    19    20
Movie1
1       0.32  0.79 -0.11 -0.20 -0.16 -0.16 -0.11  0.54 -0.11
2      -0.16  0.32 -0.11  0.20 -0.16 -0.16 -0.11 -0.11 -0.11
3       0.44 -0.12 -0.09  0.32 -0.12  0.44 -0.09  0.69 -0.09
4       0.44 -0.12  0.69 -0.16  0.44  0.44 -0.09  0.69 -0.09
5       0.69 -0.09 -0.06 -0.11 -0.09 -0.09 -0.06  1.00 -0.06
6      -0.16 -0.16 -0.11  0.20  0.32 -0.16  0.54 -0.11  0.54
7       0.44 -0.12 -0.09  0.32 -0.12  0.44 -0.09  0.69 -0.09
8      -0.12  0.44 -0.09  0.32 -0.12 -0.12 -0.09 -0.09 -0.09
9      -0.09 -0.09 -0.06  0.54 -0.09 -0.09 -0.06 -0.06  1.00
10     -0.16 -0.16 -0.11  0.60  0.32 -0.16  0.54 -0.11  0.54
11      0.32 -0.16  0.54  0.20  0.32  0.79 -0.11  0.54 -0.11
12      1.00 -0.12 -0.09 -0.16 -0.12 -0.12 -0.09  0.69 -0.09
13     -0.12  1.00 -0.09 -0.16 -0.12 -0.12 -0.09 -0.09 -0.09
14     -0.09 -0.09  1.00 -0.11  0.69  0.69 -0.06 -0.06 -0.06
15     -0.16 -0.16 -0.11  1.00 -0.16  0.32 -0.11 -0.11  0.54
16     -0.12 -0.12  0.69 -0.16  1.00  0.44  0.69 -0.09 -0.09
17     -0.12 -0.12  0.69  0.32  0.44  1.00 -0.09 -0.09 -0.09
18     -0.09 -0.09 -0.06 -0.11  0.69 -0.09  1.00 -0.06 -0.06
19      0.69 -0.09 -0.06 -0.11 -0.09 -0.09 -0.06  1.00 -0.06
20     -0.09 -0.09 -0.06  0.54 -0.09 -0.09 -0.06 -0.06  1.00
```

The above is the similarity matrix for first 20 movies and for movie with itself the pearson correlation is 1 since it is 100% similar.

[55]:
```
distances = []
for user1 in userfeatures.head(20).index:
    for user2 in userfeatures.head(20).index:
        if user1 == user2:
            continue
        _pearsonCorr, _ = pearsonr(userfeatures.loc[user1], userfeatures.
 ↪loc[user2])
        distances.append([user1, user2, _pearsonCorr])
```

[56]:
```
usersSimilarityMatrix = pd.DataFrame(distances, columns = ["User1", "User2",␣
 ↪"PearsonDistance"])
pd.pivot_table(values = 'PearsonDistance', index = "User1", columns = "User2",␣
 ↪data = usersSimilarityMatrix).fillna(1)
```

[56]:
```
User2    1     2     3     4     5     6     7     8     9     10    11    12  \
User1
1      1.00 -0.57  0.30 -0.22 -0.16 -0.08 -0.07  0.24  0.17  0.42  0.39  0.29
2     -0.57  1.00  0.48  0.66  0.26  0.22 -0.05  0.22  0.43 -0.35 -0.33  0.44
3      0.30  0.48  1.00  0.42  0.39 -0.36 -0.07  0.87  0.91 -0.38 -0.44  0.99
4     -0.22  0.66  0.42  1.00 -0.46  0.29  0.66  0.24  0.10  0.23 -0.26  0.43
```

```
5     -0.16  0.26  0.39 -0.46  1.00 -0.52 -0.79  0.42  0.72 -0.88 -0.37  0.36
6     -0.08  0.22 -0.36  0.29 -0.52  1.00  0.02 -0.75 -0.57  0.71  0.78 -0.44
7     -0.07 -0.05 -0.07  0.66 -0.79  0.02  1.00  0.09 -0.36  0.43 -0.28  0.01
8      0.24  0.22  0.87  0.24  0.42 -0.75  0.09  1.00  0.87 -0.56 -0.74  0.92
9      0.17  0.43  0.91  0.10  0.72 -0.57 -0.36  0.87  1.00 -0.69 -0.54  0.90
10     0.42 -0.35 -0.38  0.23 -0.88  0.71  0.43 -0.56 -0.69  1.00  0.74 -0.40
11     0.39 -0.33 -0.44 -0.26 -0.37  0.78 -0.28 -0.74 -0.54  0.74  1.00 -0.52
12     0.29  0.44  0.99  0.43  0.36 -0.44  0.01  0.92  0.90 -0.40 -0.52  1.00
13    -0.61  0.69 -0.17  0.67 -0.38  0.74  0.28 -0.45 -0.35  0.29  0.16 -0.20
14    -0.58 -0.33 -0.82 -0.42 -0.06 -0.14  0.11 -0.49 -0.60 -0.15 -0.13 -0.77
15    -0.37 -0.42 -0.52 -0.71  0.39 -0.54 -0.23 -0.13 -0.17 -0.51 -0.31 -0.47
16    -0.13 -0.41 -0.88 -0.56 -0.18  0.54 -0.33 -0.94 -0.76  0.39  0.73 -0.93
17    -0.60  0.25 -0.36  0.62 -0.64  0.25  0.81 -0.26 -0.54  0.27 -0.25 -0.30
18     0.66 -0.81 -0.49 -0.54 -0.37  0.33 -0.15 -0.53 -0.53  0.68  0.81 -0.52
19     0.78 -0.34  0.63 -0.32  0.41 -0.60 -0.30  0.70  0.67 -0.24 -0.15  0.63
20     0.33  0.24  0.91  0.36  0.29 -0.64  0.17  0.98  0.84 -0.41 -0.66  0.95

User2    13    14    15    16    17    18    19    20
User1
1     -0.61 -0.58 -0.37 -0.13 -0.60  0.66  0.78  0.33
2      0.69 -0.33 -0.42 -0.41  0.25 -0.81 -0.34  0.24
3     -0.17 -0.82 -0.52 -0.88 -0.36 -0.49  0.63  0.91
4      0.67 -0.42 -0.71 -0.56  0.62 -0.54 -0.32  0.36
5     -0.38 -0.06  0.39 -0.18 -0.64 -0.37  0.41  0.29
6      0.74 -0.14 -0.54  0.54  0.25  0.33 -0.60 -0.64
7      0.28  0.11 -0.23 -0.33  0.81 -0.15 -0.30  0.17
8     -0.45 -0.49 -0.13 -0.94 -0.26 -0.53  0.70  0.98
9     -0.35 -0.60 -0.17 -0.76 -0.54 -0.53  0.67  0.84
10     0.29 -0.15 -0.51  0.39  0.27  0.68 -0.24 -0.41
11     0.16 -0.13 -0.31  0.73 -0.25  0.81 -0.15 -0.66
12    -0.20 -0.77 -0.47 -0.93 -0.30 -0.52  0.63  0.95
13     1.00  0.01 -0.44  0.14  0.66 -0.36 -0.84 -0.37
14     0.01  1.00  0.85  0.55  0.43  0.04 -0.54 -0.61
15    -0.44  0.85  1.00  0.35 -0.03  0.01 -0.07 -0.30
16     0.14  0.55  0.35  1.00 -0.01  0.64 -0.47 -0.96
17     0.66  0.43 -0.03 -0.01  1.00 -0.36 -0.78 -0.22
18    -0.36  0.04  0.01  0.64 -0.36  1.00  0.20 -0.48
19    -0.84 -0.54 -0.07 -0.47 -0.78  0.20  1.00  0.69
20    -0.37 -0.61 -0.30 -0.96 -0.22 -0.48  0.69  1.00
```

The above is the similarity matrix for first 20 users and for user with itself the pearson correlation is 1 since it is 100% similar.

0.0.7 6. Content Based Recommendation system using Nearest Neighbor and Cosine Similarity

6.1 Item - Item Based similarity and recommendation

[57]:
```
itembasedmodel = NearestNeighbors(n_neighbors = 5,
                                  metric = 'cosine',
                                  algorithm = 'auto')
```

[58]:
```
itembasedmodel.fit(moviesExplodedpivotted)
```

[58]:
```
NearestNeighbors(metric='cosine')
```

[59]:
```
def getItemRecommendationNearestNeighbors(movieName):
    _movies = movies[movies['Title'].str.contains(movieName)]

    if _movies.shape[0] == 0:
        return 'No movie with given name present'

    _movieIndex = _movies.index[0]

    _, predictions = itembasedmodel.kneighbors([moviesExplodedpivotted.
 ↪iloc[_movieIndex]])

    return pd.DataFrame(movies.iloc[predictions[0]]['Title']).reset_index(drop␣
 ↪= True)
```

[60]:
```
getItemRecommendationNearestNeighbors("Titanic")
```

[60]:

	Title
0	Total Eclipse (1995)
1	Parallel Sons (1995)
2	My Name Is Joe (1998)
3	Anna and the King (1999)
4	Criminal Lovers (Les Amants Criminels) (1999)

For making item - item based recommendation we created a Nearest Neighbor model with 5 number of neighbors and fitted it with movies data which we have exploded and pivotted with 0 and 1 values for each movie with genre.

Then a function is created which accepts movie name and uses nearest neighbor model fitted above to get predictions. Then titles of those movies are found and returned as recommendations.

This is a dynamic approach which accepts movie name from user and returns recommendations.

The value of cosine similarity varies from 0 to 1 and 1 means highest similarity and 0 means low similarity

[61]:
```
movieNameInput = input("Enter movie name: ")
getItemRecommendationNearestNeighbors(movieNameInput)
```

```
Enter movie name: Toy Story
```

[61]:
```
                                      Title
0  American Tail: Fievel Goes West, An (1991)
1                       Saludos Amigos (1943)
2      Aladdin and the King of Thieves (1996)
3                    American Tail, An (1986)
4                         Toy Story 2 (1999)
```

6.2 User - User Similarity and Recommendation

[62]:
```
userbasedmodel = NearestNeighbors(n_neighbors = 6,
                                  metric = 'cosine',
                                  algorithm = 'auto')
```

[63]:
```
userfeatures.head()
```

[63]:
```
          Age  Occupation  GenderMapped  Rating  hour
UserID
1       -2.30        0.29          1.59    1.13 -0.77
2        1.97        1.24         -0.63    0.02 -0.91
3       -0.44        1.08         -0.63    0.46 -1.05
4        1.11       -0.18         -0.63    1.14 -1.19
5       -0.44        1.87         -0.63   -1.29  0.35
```

[64]:
```
userbasedmodel.fit(userfeatures)
```

[64]:
```
NearestNeighbors(metric='cosine', n_neighbors=6)
```

[65]:
```
def getUserRecommendationNearestNeighbors(userID):
    _, predictions = userbasedmodel.kneighbors([userfeatures.iloc[userID]])
    return pd.DataFrame(['User - ' + str(x) for x in predictions[0][1:]],␣
 ↪columns = ['Similar Users'])
```

[66]:
```
getUserRecommendationNearestNeighbors(userID = 50)
```

[66]:
```
  Similar Users
0    User - 469
1   User - 4157
2   User - 5113
3    User - 483
4   User - 5523
```

For making user - user based recommendation we created a Nearest Neighbor model with 6 number of neighbors (since we want 5 recommendations) and fitted it with user standard scaled data (i.e. userfeatures) which we have created in step 5.2

Then a function is created which accepts user UD and uses nearest neighbor model fitted above to get predictions. Then those users are returned as it is since index of userfeatures is same as user id

This is a dynamic approach which can also accept user ID and returns recommendations.

[67]:
```
userIDInput = int(input("Enter movie name: "))
getUserRecommendationNearestNeighbors(userIDInput)
```

```
Enter movie name: 237
```

[67]:

	Similar Users
0	User - 2374
1	User - 5837
2	User - 2333
3	User - 5108
4	User - 5050

I tried finding out similarity matrix between items and users but due to large computations my system was getting hanged. Hence, below i am attaching a smaller representation of how it will look like for both users and movies.

Hence, I am doing all calculations for top 20 movies and 20 users.

[68]:
```
distances = []
for movie1 in moviesExplodedpivotted.head(20).index:
    for movie2 in moviesExplodedpivotted.head(20).index:
        if movie1 == movie2:
            continue
        A = moviesExplodedpivotted.loc[movie1]
        B = moviesExplodedpivotted.loc[movie2]
        _cosineSimilarity = np.dot(A,B) / (np.linalg.norm(A) * np.linalg.
 ↪norm(B))
        distances.append([movie1, movie2, _cosineSimilarity])
```

[69]:
```
itemsSimilarityMatrix = pd.DataFrame(distances, columns = ["Movie1", "Movie2",␣
 ↪"CosineSimilarity"])
pd.pivot_table(values = 'CosineSimilarity', index = "Movie1", columns =␣
 ↪"Movie2", data = itemsSimilarityMatrix).fillna(1)
```

[69]:

Movie2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	\
Movie1															
1	1.00	0.33	0.41	0.41	0.58	0.00	0.41	0.41	0.00	0.00	0.33	0.41	0.82	0.00	
2	0.33	1.00	0.00	0.00	0.00	0.00	0.00	0.82	0.00	0.33	0.00	0.00	0.41	0.00	
3	0.41	0.00	1.00	0.50	0.71	0.00	1.00	0.00	0.00	0.00	0.82	0.50	0.00	0.00	
4	0.41	0.00	0.50	1.00	0.71	0.00	0.50	0.00	0.00	0.00	0.82	0.50	0.00	0.71	
5	0.58	0.00	0.71	0.71	1.00	0.00	0.71	0.00	0.00	0.00	0.58	0.71	0.00	0.00	
6	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.58	0.67	0.00	0.00	0.00	0.00	
7	0.41	0.00	1.00	0.50	0.71	0.00	1.00	0.00	0.00	0.00	0.82	0.50	0.00	0.00	
8	0.41	0.82	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.41	0.00	0.00	0.50	0.00	
9	0.00	0.00	0.00	0.00	0.00	0.58	0.00	0.00	1.00	0.58	0.00	0.00	0.00	0.00	
10	0.00	0.33	0.00	0.00	0.00	0.67	0.00	0.41	0.58	1.00	0.00	0.00	0.00	0.00	
11	0.33	0.00	0.82	0.82	0.58	0.00	0.82	0.00	0.00	0.00	1.00	0.41	0.00	0.58	
12	0.41	0.00	0.50	0.50	0.71	0.00	0.50	0.00	0.00	0.00	0.41	1.00	0.00	0.00	

```
13      0.82 0.41 0.00 0.00 0.00 0.00 0.00 0.50 0.00 0.00 0.00 0.00 1.00 0.00
14      0.00 0.00 0.00 0.71 0.00 0.00 0.00 0.00 0.00 0.00 0.58 0.00 0.00 1.00
15      0.00 0.33 0.41 0.00 0.00 0.33 0.41 0.41 0.58 0.67 0.33 0.00 0.00 0.00
16      0.00 0.00 0.00 0.50 0.00 0.41 0.00 0.00 0.00 0.41 0.41 0.00 0.00 0.71
17      0.00 0.00 0.50 0.50 0.00 0.00 0.50 0.00 0.00 0.00 0.82 0.00 0.00 0.71
18      0.00 0.00 0.00 0.00 0.00 0.58 0.00 0.00 0.00 0.58 0.00 0.00 0.00 0.00
19      0.58 0.00 0.71 0.71 1.00 0.00 0.71 0.00 0.00 0.00 0.58 0.71 0.00 0.00
20      0.00 0.00 0.00 0.00 0.00 0.58 0.00 0.00 1.00 0.58 0.00 0.00 0.00 0.00

Movie2   15   16   17   18   19   20
Movie1
1       0.00 0.00 0.00 0.00 0.58 0.00
2       0.33 0.00 0.00 0.00 0.00 0.00
3       0.41 0.00 0.50 0.00 0.71 0.00
4       0.00 0.50 0.50 0.00 0.71 0.00
5       0.00 0.00 0.00 0.00 1.00 0.00
6       0.33 0.41 0.00 0.58 0.00 0.58
7       0.41 0.00 0.50 0.00 0.71 0.00
8       0.41 0.00 0.00 0.00 0.00 0.00
9       0.58 0.00 0.00 0.00 0.00 1.00
10      0.67 0.41 0.00 0.58 0.00 0.58
11      0.33 0.41 0.82 0.00 0.58 0.00
12      0.00 0.00 0.00 0.00 0.71 0.00
13      0.00 0.00 0.00 0.00 0.00 0.00
14      0.00 0.71 0.71 0.00 0.00 0.00
15      1.00 0.00 0.41 0.00 0.00 0.58
16      0.00 1.00 0.50 0.71 0.00 0.00
17      0.41 0.50 1.00 0.00 0.00 0.00
18      0.00 0.71 0.00 1.00 0.00 0.00
19      0.00 0.00 0.00 0.00 1.00 0.00
20      0.58 0.00 0.00 0.00 0.00 1.00
```

The above is the similarity matrix for first 20 movies and for movie with itself the Cosine similarity is 1 since it is 100% similar.

[70]:
```
distances = []
for user1 in userfeatures.head(20).index:
    for user2 in userfeatures.head(20).index:
        if user1 == user2:
            continue
        A = userfeatures.loc[user1]
        B = userfeatures.loc[user2]
        _cosineSimilarity = np.dot(A,B) / (np.linalg.norm(A) * np.linalg.
 ↪norm(B))
        distances.append([user1, user2, _cosineSimilarity])
```

[71]:
```
usersSimilarityMatrix = pd.DataFrame(distances, columns = ["User1", "User2",␣
↪"CosineSimilarity"])
pd.pivot_table(values = 'CosineSimilarity', index = "User1", columns = "User2",␣
↪data = usersSimilarityMatrix).fillna(1)
```

[71]:

User2	1	2	3	4	5	6	7	8	9	10	11	12
User1												
1	1.00	-0.55	0.30	-0.22	-0.16	-0.06	-0.07	0.24	0.17	0.40	0.37	0.28
2	-0.55	1.00	0.41	0.64	0.24	0.36	-0.05	0.18	0.40	-0.24	-0.40	0.30
3	0.30	0.41	1.00	0.41	0.39	-0.35	-0.07	0.87	0.91	-0.40	-0.36	0.98
4	-0.22	0.64	0.41	1.00	-0.46	0.23	0.66	0.23	0.09	0.23	-0.26	0.39
5	-0.16	0.24	0.39	-0.46	1.00	-0.36	-0.79	0.42	0.72	-0.86	-0.33	0.35
6	-0.06	0.36	-0.35	0.23	-0.36	1.00	0.00	-0.57	-0.39	0.66	0.22	-0.52
7	-0.07	-0.05	-0.07	0.66	-0.79	0.00	1.00	0.09	-0.36	0.41	-0.26	0.01
8	0.24	0.18	0.87	0.23	0.42	-0.57	0.09	1.00	0.87	-0.56	-0.66	0.89
9	0.17	0.40	0.91	0.09	0.72	-0.39	-0.36	0.87	1.00	-0.67	-0.50	0.86
10	0.40	-0.24	-0.40	0.23	-0.86	0.66	0.41	-0.56	-0.67	1.00	0.56	-0.45
11	0.37	-0.40	-0.36	-0.26	-0.33	0.22	-0.26	-0.66	-0.50	0.56	1.00	-0.35
12	0.28	0.30	0.98	0.39	0.35	-0.52	0.01	0.89	0.86	-0.45	-0.35	1.00
13	-0.52	0.40	-0.06	0.54	-0.31	0.02	0.24	-0.32	-0.28	0.08	0.31	0.01
14	-0.58	-0.35	-0.79	-0.42	-0.06	-0.17	0.11	-0.48	-0.60	-0.17	-0.09	-0.69
15	-0.36	-0.42	-0.50	-0.71	0.39	-0.42	-0.23	-0.12	-0.17	-0.51	-0.26	-0.42
16	-0.13	-0.39	-0.87	-0.56	-0.18	0.36	-0.33	-0.93	-0.76	0.37	0.69	-0.87
17	-0.59	0.29	-0.38	0.62	-0.64	0.31	0.79	-0.28	-0.53	0.31	-0.29	-0.35
18	0.66	-0.78	-0.48	-0.54	-0.37	0.20	-0.15	-0.52	-0.53	0.64	0.77	-0.48
19	0.58	-0.44	0.56	-0.27	0.32	-0.80	-0.21	0.58	0.52	-0.36	0.13	0.66
20	0.32	0.26	0.88	0.36	0.28	-0.33	0.17	0.96	0.83	-0.36	-0.65	0.85

User2	13	14	15	16	17	18	19	20
User1								
1	-0.52	-0.58	-0.36	-0.13	-0.59	0.66	0.58	0.32
2	0.40	-0.35	-0.42	-0.39	0.29	-0.78	-0.44	0.26
3	-0.06	-0.79	-0.50	-0.87	-0.38	-0.48	0.56	0.88
4	0.54	-0.42	-0.71	-0.56	0.62	-0.54	-0.27	0.36
5	-0.31	-0.06	0.39	-0.18	-0.64	-0.37	0.32	0.28
6	0.02	-0.17	-0.42	0.36	0.31	0.20	-0.80	-0.33
7	0.24	0.11	-0.23	-0.33	0.79	-0.15	-0.21	0.17
8	-0.32	-0.48	-0.12	-0.93	-0.28	-0.52	0.58	0.96
9	-0.28	-0.60	-0.17	-0.76	-0.53	-0.53	0.52	0.83
10	0.08	-0.17	-0.51	0.37	0.31	0.64	-0.36	-0.36
11	0.31	-0.09	-0.26	0.69	-0.29	0.77	0.13	-0.65
12	0.01	-0.69	-0.42	-0.87	-0.35	-0.48	0.66	0.85
13	1.00	0.06	-0.32	0.12	0.45	-0.29	-0.18	-0.38
14	0.06	1.00	0.85	0.55	0.40	0.04	-0.33	-0.61
15	-0.32	0.85	1.00	0.35	-0.05	0.01	0.00	-0.30
16	0.12	0.55	0.35	1.00	-0.01	0.64	-0.35	-0.96
17	0.45	0.40	-0.05	-0.01	1.00	-0.35	-0.70	-0.20

```
18    -0.29  0.04  0.01  0.64 -0.35  1.00  0.16 -0.48
19    -0.18 -0.33  0.00 -0.35 -0.70  0.16  1.00  0.43
20    -0.38 -0.61 -0.30 -0.96 -0.20 -0.48  0.43  1.00
```

The above is the similarity matrix for first 20 users and for user with itself the cosine similarty is 1 since it is 100% similar.

[72]:
```
testMatrix = [[1 , 0],
              [3 , 7]]
sparseMatrix = sparse.csr_matrix(testMatrix)
print(sparseMatrix)
```

```
  (0, 0)        1
  (1, 0)        3
  (1, 1)        7
```

To further enhance processing we can use sparse representation as illustrated above where instead of working on whole data we can use data where value is not zero as indicated above as for row index and column index as 0 the value is 1, for 1 and 0 value is 3 and it can help us to save lot of time on calculation.

0.0.8 7. Matrix Factorization (Combining user and items together)

[73]:
```
ratings.head()
```

[73]:

	UserID	MovieID	Rating	Timestamp	hour
0	1	1193	5	978300760	3
1	1	661	3	978302109	4
2	1	914	3	978301968	4
3	1	3408	4	978300275	3
4	1	2355	5	978824291	5

For matrix factorization we are using a library called cmrfec which required data in a particular fashion as the column name needs to be UserId, ItemId, Rating so we extracted columns UserID, MovieID, Rating and renamed them to required columns

[74]:
```
mfratingsFormat = ratings[["UserID", "MovieID", "Rating"]]
mfratingsFormat.columns = ['UserId', 'ItemId', 'Rating']
mfratingsFormat.head(3)
```

[74]:

	UserId	ItemId	Rating
0	1	1193	5
1	1	661	3
2	1	914	3

For fitting the algorithm we created a model named as mfmodel and we passed following parameters inside

- method is als which means what is the optimization method

- k or the number of dimensions required is 4 in our case
- lambda_ is set to 0.1 which denotes Regularization parameter
- user_bias is set to False as it means whether to add user/row biases (intercepts) to the model
- item_bias is set to False as it means whether to add item/column biases (intercepts) to the model
- verbose is set to False so that nothing gets printed on screen

[75]:
```
mfmodel = CMF(method = "als",
              k = 4,
              lambda_ = 0.1,
              user_bias = False,
              item_bias = False,
              verbose = False)

mfmodel.fit(mfratingsFormat)
```

[75]:
```
Collective matrix factorization model
(explicit-feedback variant)
```

[76]:
```
ratingsPivotted = ratings.pivot(index = 'UserID', columns ='MovieID', values =␣
↪'Rating').fillna(0)
ratingsPivotted.head()
```

[76]:
```
MovieID  1     2     3     4     5     6     7     8     9     10    …  \
UserID                                                                …
1        5.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  …
2        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  …
3        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  …
4        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  …
5        0.00  0.00  0.00  0.00  0.00  2.00  0.00  0.00  0.00  0.00  …

MovieID  3943  3944  3945  3946  3947  3948  3949  3950  3951  3952
UserID
1        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
2        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
3        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
4        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
5        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

[5 rows x 3706 columns]
```

This above dataframe represents relation between

[77]:
```
ratingMatrix = np.dot(mfmodel.A_, mfmodel.B_.T) + mfmodel.glob_mean_
mseError = mean_squared_error(ratingsPivotted.values[ratingsPivotted > 0],␣
↪ratingMatrix[ratingsPivotted > 0]) ** 0.5
print(f"The mean squared error is: {mseError}")
```

```
The mean squared error is: 1.3451500956582787
```

[78]:
```
ratingMatrix = np.dot(mfmodel.A_, mfmodel.B_.T) + mfmodel.glob_mean_
mapeError = mean_absolute_percentage_error(ratingsPivotted.
 ↪values[ratingsPivotted > 0], ratingMatrix[ratingsPivotted > 0])
print(f"The mean absolute percentage error is: {mapeError}")
```

```
The mean absolute percentage error is: 0.3781874875455212
```

For finding the mean squared error and mean absolute percentage error we use pivotting on ratings dataframe and then on calculation we can see that they are 1.34 and 0.37 which is good but they can be too less to provide real time recommendations.

Also, the benefit of using this is that we get embeddings of user and items so instead of taking whole feature we can use this.

Here, mfmodel.A_ represents User Embeddings and mfmodel.B_ represents Item embeddings

On the basis of approach, Collaborative Filtering methods can be classified into item - item based and classification based.

0.0.9 8. Using embeddings to build similarity (making use of cosine similarity)

We can use these embeddings to find similarity between items and they work as embeddings for us as they exhibit properties of user/item but we don't know how they are made .

We will use Cosine similarity to find nearest distance

Initally, we will do first for users.

[79]:
```
def getUserRecommendationCosineSimilaritywithEmbeddings(userID,␣
 ↪noofRecommendation = 10):
    distances = []
    for user, val in enumerate(mfmodel.A_):
        if userID == user:
            continue
        A = mfmodel.A_[userID]
        B = mfmodel.A_[user]
        _cosineSimilarity = np.dot(A,B) / (np.linalg.norm(A) * np.linalg.
 ↪norm(B))
        distances.append([user, _cosineSimilarity])

    userRanking = pd.DataFrame(distances, columns = ['SimilarUser',␣
 ↪'CosineSimilarity'])
    userRanking.sort_values('CosineSimilarity', ascending = False, inplace =␣
 ↪True)
    userRanking.reset_index(inplace = True, drop = True)
    return userRanking.head(noofRecommendation)
```

[80]:
```
getUserRecommendationCosineSimilaritywithEmbeddings(50, noofRecommendation = 5)
```

[80]:

	SimilarUser	CosineSimilarity
0	2135	1.00
1	3149	1.00
2	5643	1.00
3	2373	0.99
4	5730	0.99

To design the item - item similarity we need to some processing as the cmfrec stores movie data in order they are in ratings data so we first take movie name, find if it is present in ratings data and if present at what index it is present in (using unique values as that is what cmfrec is using as per their documentation) then use cmfrec to find its embedding and then calculate its similarity from other movies to recommend k top movies

[81]:
```
def getItemRecommendationCosineSimilaritywithEmbeddings(movieName,␣
↪noofRecommendation = 5):
    _movies = movies[movies['Title'].str.contains(movieName)]

    if _movies.shape[0] == 0:
        return 'No movie with this name available in the database'

    _movieIndex = _movies['Movie ID'].iloc[0]
    _ratings = ratings[ratings['MovieID'] == _movieIndex]
    ratingsUnique = list(ratings['MovieID'].unique())

    if _ratings.shape[0] == 0:
        return "This movie has no ratings yet in ratings dataframe"

    A = mfmodel.B_[ratingsUnique.index(_movieIndex)]

    distances = []
    for index, moviesIndex in enumerate(mfmodel.item_mapping_):
        if moviesIndex == _movieIndex:
            continue
        B = mfmodel.B_[index]
        _cosineSimilarity = np.dot(A,B) / (np.linalg.norm(A) * np.linalg.
↪norm(B))
        distances.append([moviesIndex, _cosineSimilarity])

    moviesRanked = pd.DataFrame(distances, columns = ['SimilarMovies',␣
↪'CosineSimilarity'])
    moviesRanked['SimilarMovies'] = moviesRanked['SimilarMovies'].apply(lambda␣
↪x : movies[movies['Movie ID'] == x]['Title'].iloc[0])
    moviesRanked.sort_values('CosineSimilarity', ascending = False, inplace =␣
↪True)
    moviesRanked.reset_index(inplace = True, drop = True)
    return moviesRanked.head(noofRecommendation)
```

[82]:
```
getItemRecommendationCosineSimilaritywithEmbeddings("Titanic")
```

[82]:

	SimilarMovies	CosineSimilarity
0	Ghosts of Mississippi (1996)	1.00
1	Pretty Woman (1990)	0.99
2	Ever After: A Cinderella Story (1998)	0.99
3	Ghost (1990)	0.99
4	Speed (1994)	0.99

Hence, we can see that results are little better when we used embedding but it can be better if embeddings size is bigger as lot of data loss is happening here

0.0.10 9. Visualizing User Embeddings given by Matrix Factorization

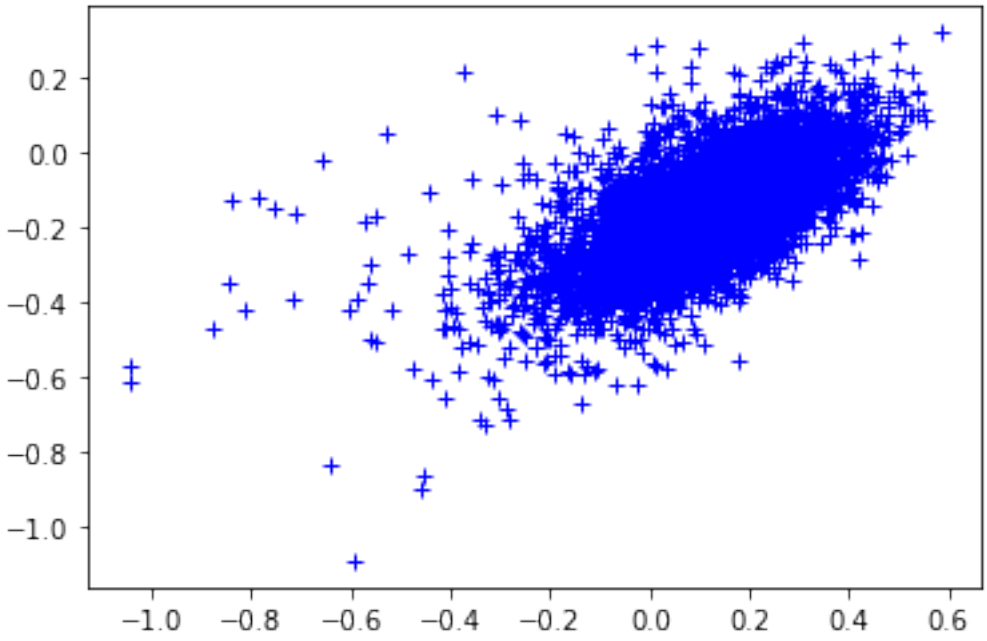
Using 2 dimensions of embeddings for better visualizations

[83]:
```
mfmodelwith2dimensions = CMF(method = "als",
                             k = 2,
                             lambda_ = 0.1,
                             user_bias = False,
                             item_bias = False,
                             verbose = False)

mfmodelwith2dimensions.fit(mfratingsFormat)
```

[83]:
```
Collective matrix factorization model
(explicit-feedback variant)
```

[84]:
```
plt.plot([x[0] for x in mfmodelwith2dimensions.A_],
         [x[1] for x in mfmodelwith2dimensions.A_],
         'b+')
plt.show()
```

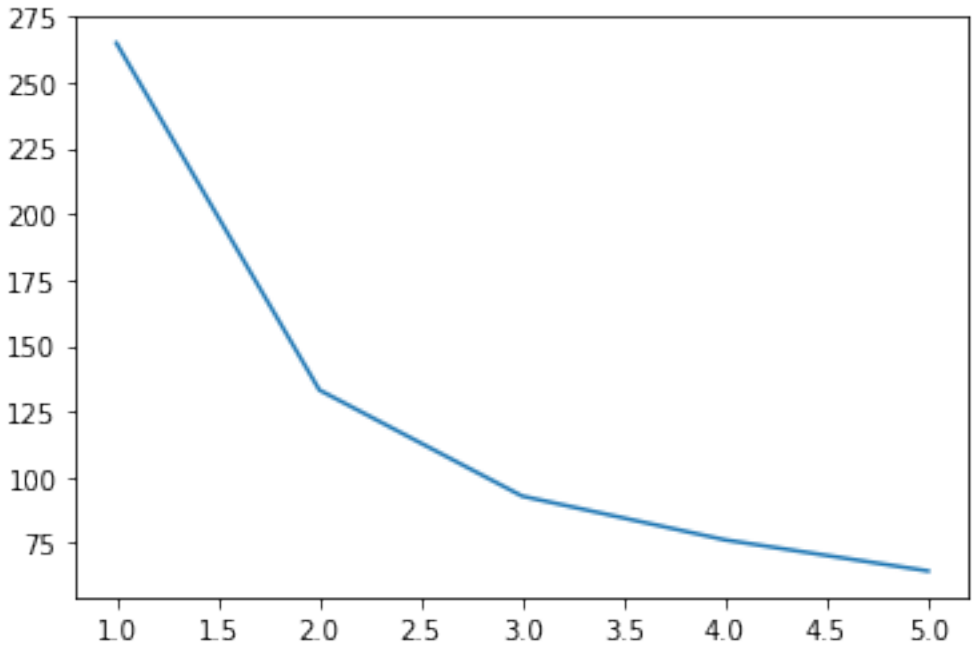


From above results we cannot see anything hence, let us try to use clustering and see if it can give some results.

[85]:
```
wcss_scores = []

for k in range(1, 6):
    kmeansmodel = KMeans(n_clusters = k)
    kmeansmodel.fit(mfmodelwith2dimensions.A_)
    wcss_scores.append(kmeansmodel.inertia_)

plt.plot(np.arange(1, 6, 1), wcss_scores)
plt.show()
```

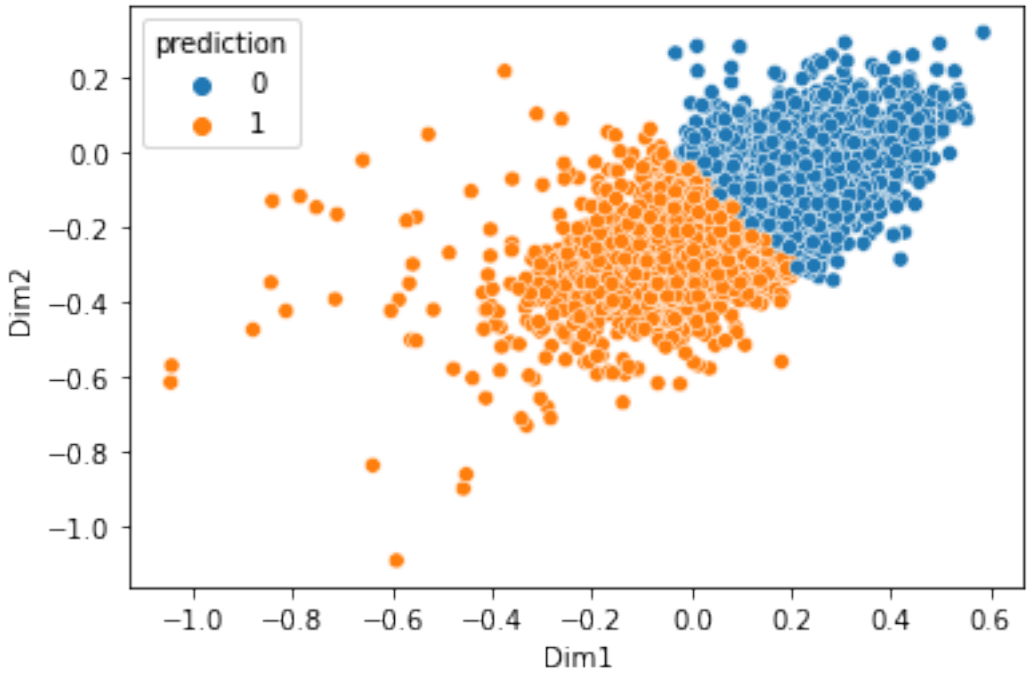


We can see that using K means elbow occurred at K = 2 hence, there are mainly 2 clusters for all the users present in our data.

[86]:
```
kmeansmodel = KMeans(n_clusters = 2)
kmeansmodel.fit(mfmodelwith2dimensions.A_)

userMFdf = pd.DataFrame(mfmodelwith2dimensions.A_, columns = ['Dim1', 'Dim2'])
userMFdf['prediction'] = kmeansmodel.predict(mfmodelwith2dimensions.A_)

sns.scatterplot(x = 'Dim1', y = 'Dim2', hue = 'prediction', data = userMFdf)
plt.show()
```



We can see that there are majorly 2 types of users present in our dataframe. Maybe one are who watch more movies and give high ratings and one maybe less watching movies and give less ratings.

0.0.11 10. Creating a recommendation system based on what User likes

We can build a real time scenario where we defined a new user (not in DB) with some movies he has watched and see how recommendation system works in real time.

Let UserID be 9999 with following User Attributes - Age : 25 (25 - 34 years old) - Occupation : 12 (programmer) - Gender : 0 (i.e. Male)

and he has watched movies with ID - 1 (Toy Story) - 3949 (Requiem for a Dream) - 5 (Father of the Bride Part II) - 770 (Costa Brava)

[87]:
```
inputUserdf = pd.DataFrame(columns = ['Age', 'Occupation', 'GenderMapped'])
inputUserdf['Age'] = [25]
inputUserdf['Occupation'] = [12]
inputUserdf['GenderMapped'] = [0]
inputUserdf
```

[87]:

	Age	Occupation	GenderMapped
0	25	12	0

[88]:
```
inputRatingdf = pd.DataFrame(columns = ['UserID', 'MovieID', 'Rating', 'Hour'])
inputRatingdf['UserID'] = [9999, 9999, 9999, 9999]
inputRatingdf['MovieID'] = [1, 3949, 5, 770]
```

```
inputRatingdf['Rating'] = [4.25, 2.1, 1.09, 3.87]
inputRatingdf['Hour'] = [6, 6, 8, 7]
inputRatingdf
```

[88]:

	UserID	MovieID	Rating	Hour
0	9999	1	4.25	6
1	9999	3949	2.10	6
2	9999	5	1.09	8
3	9999	770	3.87	7

Now, let us recommend movies based on the movies he had watched.

[89]:
```
recommendedMovies = set()

for mid in inputRatingdf['MovieID'].unique():
    _movieName = movies[movies['Movie ID'] == mid]['Title'].iloc[0].
 ↪split('(')[0]
    _recommendationIndivudal =␣
 ↪getItemRecommendationNearestNeighbors(_movieName).values.tolist()
    for _r in _recommendationIndivudal:
        recommendedMovies.add(_r[0])

print("Recommended movies based on what he has watched is ")
pd.DataFrame(list(recommendedMovies), columns = ['Recommendation'])
```

```
Recommended movies based on what he has watched is
```

[89]:

	Recommendation
0	Aladdin and the King of Thieves (1996)
1	Flirt (1995)
2	Cool Runnings (1993)
3	Wayne's World 2 (1993)
4	American Tail, An (1986)
5	Saludos Amigos (1943)
6	Dingo (1992)
7	Associate, The (L'Associe)(1982)
8	Without Limits (1998)
9	Death Becomes Her (1992)
10	American Tail: Fievel Goes West, An (1991)
11	Basquiat (1996)
12	Midnight Express (1978)
13	Toy Story 2 (1999)
14	White Men Can't Jump (1992)

Also, we can find user similarity using cosine similarity with other users

[90]:
```
inputuserfeatures = inputUserdf.iloc[0].values.tolist()
inputuserfeatures.append(inputRatingdf['Rating'].mean())
inputuserfeatures.append(inputRatingdf['Hour'].median())
inputuserfeatures
```

[90]:
```
[25, 12, 0, 2.8274999999999997, 6.5]
```

[91]:
```
inputuserfeaturesscaled = userscaler.transform([inputuserfeatures])
A = list(inputuserfeaturesscaled[0])

distances = []
for user in userfeatures.index:
    B = userfeatures.loc[user]
    _cosineSimilarity = np.dot(A , B) / (np.linalg.norm(A) * np.linalg.norm(B))
    distances.append([user, _cosineSimilarity])

userRanking = pd.DataFrame(distances, columns = ['SimilarUser',␣
 ↪'CosineSimilarity'])
userRanking.sort_values('CosineSimilarity', ascending = False, inplace = True)
userRanking.reset_index(inplace = True, drop = True)
userRanking.head(5)
```

[91]:

	SimilarUser	CosineSimilarity
0	1193	1.00
1	2796	0.99
2	1968	0.99
3	4855	0.99
4	3012	0.99

We can see that the user 9999 is similar to User

- 1193
- 2796
- 1968
- 4855
- 3012

Hence, movies liked by these users can be recommended to User 9999 and same matrix factorization can also be trained for the same.