Problem Statement:

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.

Data Description:

RATINGS FILE DESCRIPTION

==========

All ratings are contained in the file "ratings.dat" and are in the following format:

UserID::MovieID::Rating::Timestamp

- 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
- Each user has at least 20 ratings

USERS FILE DESCRIPTION

==========

User information is in the file "users.dat" and is in the following format:

UserID::Gender::Age::Occupation::Zip-code

All demographic information is provided voluntarily by the users and is not checked for accuracy.
Only users who have provided some demographic information are included in this data set.

- 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"

MOVIES FILE DESCRIPTION

==========

Movie information is in the file "movies.dat" and is in the following format:

MovieID::Title::Genres

- 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

Our Approach:

In this project, we'll be building a recommender system that is going to recommend movies to a user based on their preferences as well as the choices of other users who are similar to them.

What is a Recommender System?

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

Types of Recommender Systems -

Recommender systems usually make use of either or both *Collaborative Filtering* and *Content-based Filtering* techniques.

Collaborative Filtering

Collaborative filtering is based on the assumption that people who agreed in the past will agree in the future, and that they will like similar kinds of items as they liked in the past. The system generates recommendations using only information about rating profiles for different users or items. By locating peer users/items with a rating history similar to the current user or item, they generate recommendations using this neighborhood.

Content-based Filtering

Content-based filtering methods are based on a description of the item and a profile of the user's preferences. These methods are best suited to situations where there is known data on an item (name, location, description, etc.), but not on the user. Content-based recommenders treat recommendation as a user-specific classification problem and learn a classifier for the user's likes and dislikes based on an item's features.

Importing required libraries -

```
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import sparse
from scipy.stats import pearsonr
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.neighbors import NearestNeighbors
import warnings
import keras
from tensorflow.keras.optimizers import Adam
from keras.layers import Input, Embedding, Flatten
from keras.layers import dot
from pylab import rcParams
```

Configuring the notebook -

```
warnings.simplefilter('ignore')
pd.set_option("display.max_columns", None)
pd.options.display.float_format = '{:.2f}'.format
sns.set_style('white')
```

Reading the data files -

```
movies = pd.read_fwf('../content/zee-movies.dat', encoding='ISO-8859-1')
ratings = pd.read_fwf('../content/zee-ratings.dat', encoding='ISO-8859-1')
users = pd.read_fwf('../content/zee-users.dat', encoding='ISO-8859-1')
```

Data Formatting -

```
movies.head()
```

	Movie ID::Title::Genres	Unnamed: 1	Unnamed: 2
0	1::Toy Story (1995)::Animation|Children's|Comedy	NaN	NaN
1	2::Jumanji (1995)::Adventure|Children's|Fantasy	NaN	NaN
2	3::Grumpier Old Men (1995)::Comedy|Romance	NaN	NaN
3	4::Waiting to Exhale (1995)::Comedy|Drama	NaN	NaN
4	5::Father of the Bride Part II (1995)::Comedy	NaN	NaN

```
movies.drop(columns=['Unnamed: 1', 'Unnamed: 2'], axis=1, inplace=True)
```

```
delimiter = '::'
movies = movies['Movie ID::Title::Genres'].str.split(delimiter, expand=True)
movies.columns = ['Movie ID', 'Title', 'Genres']
```

```
movies.rename(columns={'Movie ID':'MovieID'}, inplace=True)
```

```
movies.sample(5)
```

	MovieID	Title	Genres
2557	2626	Edge of Seventeen (1998)	Comedy|Drama|Romance
3324	3393	Date with an Angel (1987)	Comedy|Fantasy
608	612	Pallbearer, The (1996)	Comedy
387	391	Jason's Lyric (1994)	Crime|Drama
253	256	Junior (1994)	Comedy|Sci-Fi

```
ratings = ratings['UserID::MovieID::Rating::Timestamp'].str.split(delimiter, expand=True)
ratings.columns = ['UserID', 'MovieID', 'Rating', 'Timestamp']
```

```
ratings.head()
```

	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

```
users = users['UserID::Gender::Age::Occupation::Zip-code'].str.split(delimiter, expand=True)
users.columns = ['UserID', 'Gender', 'Age', 'Occupation', 'Zip-code']
```

```
users.replace({'Age':{'1':  "Under 18",
                      '18':  "18-24",
                      '25':  "25-34",
                      '35':  "35-44",
                      '45':  "45-49",
                      '50':  "50-55",
                      '56':  "56 Above"}}, inplace=True)
```

```
users.replace({'Occupation':{'0': "other",
                             '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"}}, inplace=True)
```

```
users.head()
```

	UserID	Gender	Age	Occupation	Zip-code
0	1	F	Under 18	k-12 student	48067
1	2	M	56 Above	self-employed	70072
2	3	M	25-34	scientist	55117
3	4	M	45-49	executive/managerial	02460
4	5	M	25-34	writer	55455

Merging the dataframes -

```
df_1 = pd.merge(movies, ratings, how='inner', on='MovieID')
df_1.head()
```

	MovieID	Title	Genres	UserID	Rating	Timestamp
0	1	Toy Story (1995)	Animation|Children's|Comedy	1	5	978824268
1	1	Toy Story (1995)	Animation|Children's|Comedy	6	4	978237008
2	1	Toy Story (1995)	Animation|Children's|Comedy	8	4	978233496
3	1	Toy Story (1995)	Animation|Children's|Comedy	9	5	978225952
4	1	Toy Story (1995)	Animation|Children's|Comedy	10	5	978226474

```
df_2 = pd.merge(df_1, users, how='inner', on='UserID')
df_2.head()
```

	MovieID	Title	Genres	UserID	Rating	Timestamp	Genc
0	1	Toy Story (1995)	Animation|Children's|Comedy	1	5	978824268	
1	48	Pocahontas (1995)	Animation|Children's|Musical|Romance	1	5	978824351	
2	150	Apollo 13 (1995)	Drama	1	5	978301777	
3	260	Star Wars: Episode IV - A New Hope (1977)	Action|Adventure|Fantas	1	4	978300760	

```
data = df_2.copy(deep=True)
data.sample(10)
```

	MovieID	Title	Genres	UserID	Rating	Timestamp	Gender	Age
176192	2336	Elizabeth (1998)	Drama	768	5	975452845	M	25-34
99150	3148	Cider House Rules, The (1999)	Drama	1088	4	983744009	F	Under 18
161326	786	Eraser (1996)	Action|Thriller	578	3	975978555	M	18-24
100363	3039	Trading Places (1983)	Comedy	1100	3	974925031	M	25-34
132349	2918	Ferris Bueller's Day Off (1986)	Comedy	1066	5	974945238	M	45-49

Performing EDA -

Shape of the dataset -

```
print("No. of rows: ", data.shape[0])
print("No. of columns: ", data.shape[1])
```

```
No. of rows:  176218
No. of columns:  10
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 176218 entries, 0 to 176217
Data columns (total 10 columns):
 #   Column      Non-Null Count   Dtype 
---  ------      --------------   ----- 
 0   MovieID     176218 non-null  object
 1   Title       176218 non-null  object
 2   Genres      175563 non-null  object
```

```
 3   UserID      176218 non-null  object
 4   Rating      176218 non-null  object
 5   Timestamp   176218 non-null  object
 6   Gender      176218 non-null  object
 7   Age         176218 non-null  object
 8   Occupation  176218 non-null  object
 9   Zip-code    176218 non-null  object
dtypes: object(10)
memory usage: 14.8+ MB
```

Feature Engineering -

```
data['Rating'].unique()
```

```
array(['5', '4', '3', '2', '1'], dtype=object)
```

```
data.replace({'Rating':{'5:':'5'}}, inplace=True)
```

```
data['Rating'] = data['Rating'].astype('int32')
```

```
data['Datetime'] = pd.to_datetime(data['Timestamp'],
                                  unit='s')
```

```
data['ReleaseYear'] = data['Title'].str.rsplit(' ', 1).str[1]
data['ReleaseYear'] = data['ReleaseYear'].str.lstrip("(").str.rstrip(")")
```

```
data['ReleaseYear'].unique()
```

```
array(['1995', '1977', '1993', '1992', '1937', '1991', '1996', '1964',
       '1939', '1958', '1950', '1941', '1965', '1982', '1975', '1987',
       '1962', '1989', '1985', '1959', '1997', '1998', '1988', '1942',
       '1947', '1999', '1980', '1983', '1986', '1990', '2000', '1964):',
       '1994', '1978', '1961', '1984', '1972', '1976', '1981', '1973',
       '1974', '1940', 'Bo', '1952', '1954', '1953', '1944', '1968',
       '1957', '1946', '1949', '1951', '1963', '1971', '1979', '1967',
       '1966', '1948', '1933', '1970', '1969', '1930', '1955', '1956', '',
       '1920', '1925', '1938', '195', '1960', '1935', '1932', '1931',
       '1945', '1943', '1981):', '1934', '1936', '1929', 'the', '1926',
       'Arta', 'B', '1927', '19', '1922', 'Polar', '1919', '1921', "d'A",
       '1923', '1989):', '1928', '1995):', 'prendront'], dtype=object)
```

```
data['ReleaseYear'].replace(['1964):', '1981):', '1989):', '1995):'],
                            ['1964', '1981', '1989', '1995'], inplace=True)
```

```
idx_val = data[(data['ReleaseYear']=='Bo') |
               (data['ReleaseYear']=='195') |
               (data['ReleaseYear']=='') |
               (data['ReleaseYear']=='the') |
               (data['ReleaseYear']=='Arta') |
               (data['ReleaseYear']=='B') |
               (data['ReleaseYear']=='19') |
               (data['ReleaseYear']=='Polar') |
               (data['ReleaseYear']=="d'A") |
               (data['ReleaseYear']=='prendront') |
               (data['ReleaseYear']=='1')].index
data.drop(index=idx_val, inplace=True)
```

```
data['ReleaseYear'] = data['ReleaseYear'].astype('int32')
```

```
data['Title'] = data['Title'].str.rsplit(' ', 1).str[0]
```

```
bins = [1919, 1929, 1939, 1949, 1959, 1969, 1979, 1989, 2000]
labels = ['20s', '30s', '40s', '50s', '60s', '70s', '80s', '90s']
data['ReleaseDec'] = pd.cut(data['ReleaseYear'], bins=bins, labels=labels)
```

```
data.sample(5)
```

	MovieID	Title	Genres	UserID	Rating	Timestamp	Gender	Age
174056	3949	Requiem for a Dream	Drama	747	5	975465023	M	18-24
40298	1373	Star Trek V: The Final Frontier	Action|Adventure|Sci-Fi	516	3	976281005	F	56 Above
139919	904	Rear Window	Mystery|Thriller	824	4	975377094	M	35-44
90063	1286	Somewhere in Time	Drama|Romance	1010	4	975229920	M	25-34
45387	544	Striking Distance	Action	549	3	976118158	M	25-34

Data Cleaning -

Checking for null values -

```
data.isna().sum()
```

```
MovieID          0
Title            0
Genres         258
UserID           0
Rating           0
Timestamp        0
Gender           0
Age              0
Occupation       0
Zip-code         0
Datetime         0
ReleaseYear      0
ReleaseDec       5
dtype: int64
```

Checking for duplicate rows -

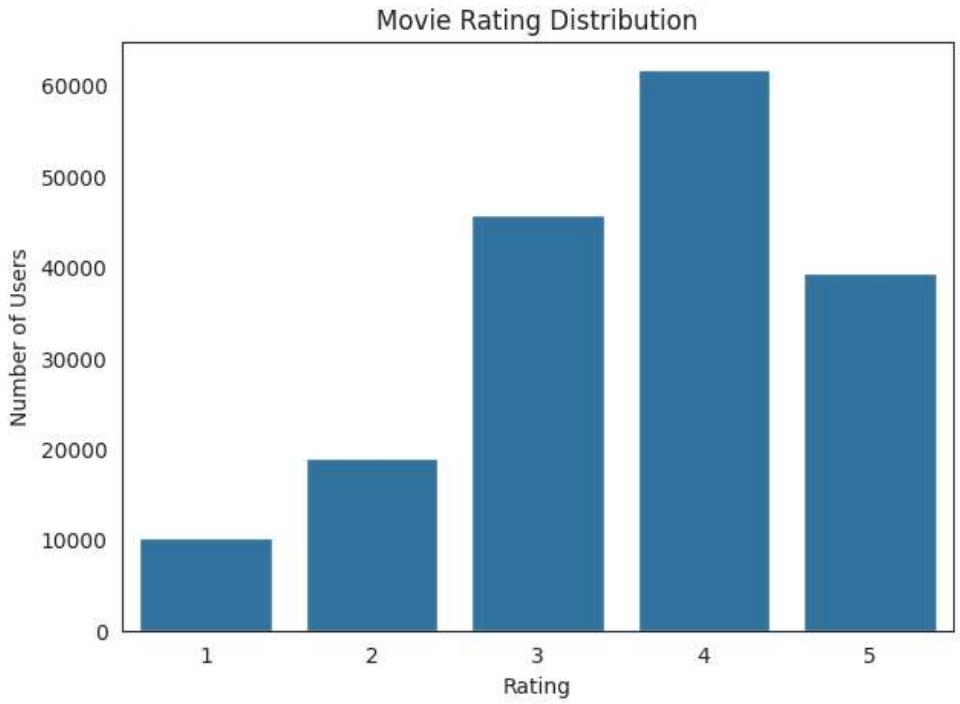
```
duplicate_rows = data[data.duplicated()]
print("No. of duplicate rows: ", duplicate_rows.shape[0])
```

```
No. of duplicate rows:  0
```

Data Visualization -

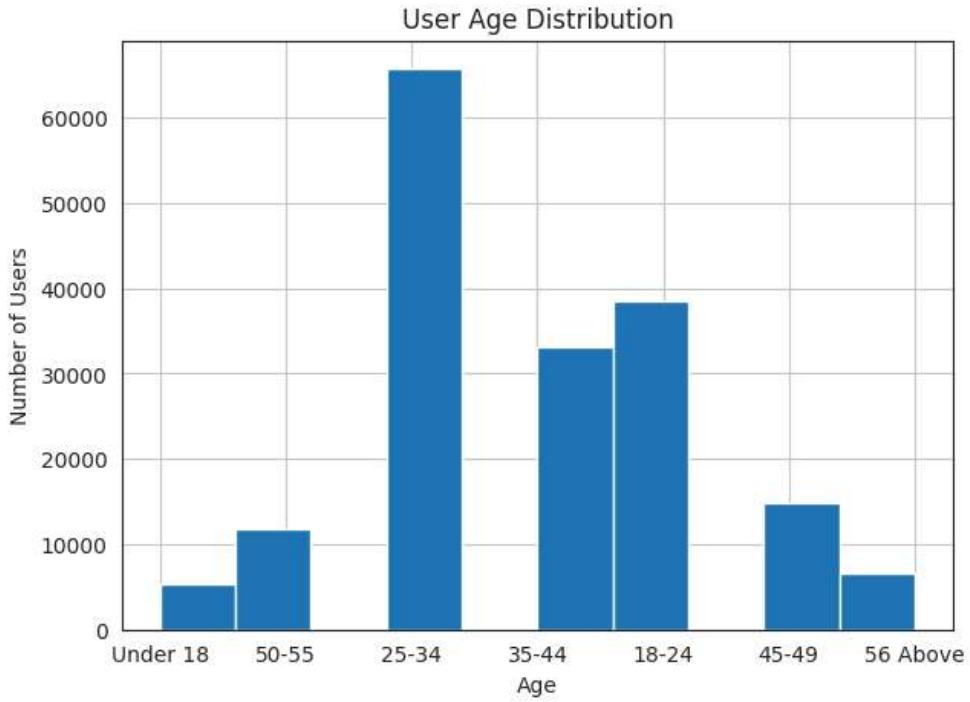
Distribution of Movie Ratings -

```
plt.figure(figsize=(7, 5))
sns.countplot(x='Rating', data=data)
plt.title('Movie Rating Distribution')
plt.xlabel('Rating')
plt.ylabel('Number of Users')
plt.show()
```



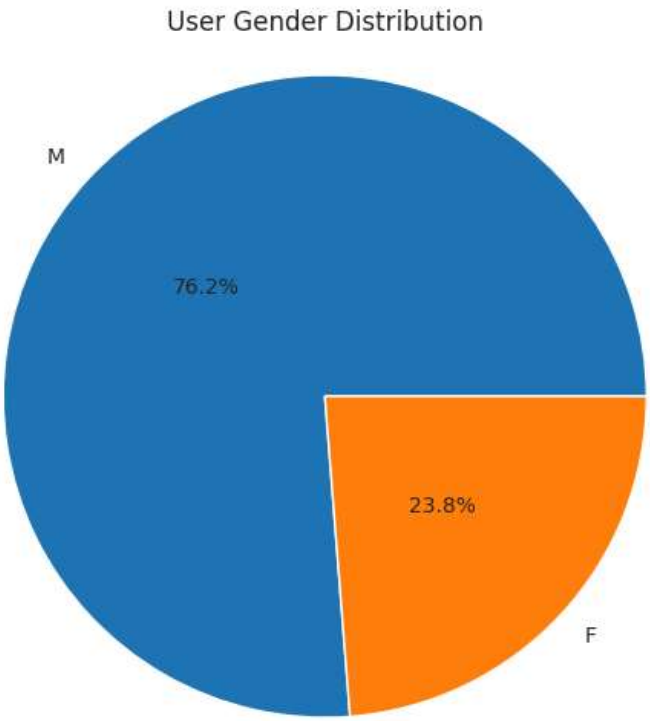
Distribution by Age -

```
data['Age'].hist(figsize=(7, 5))
plt.title('User Age Distribution')
plt.xlabel('Age')
plt.ylabel('Number of Users')
plt.show()
```



Distribution by Gender -

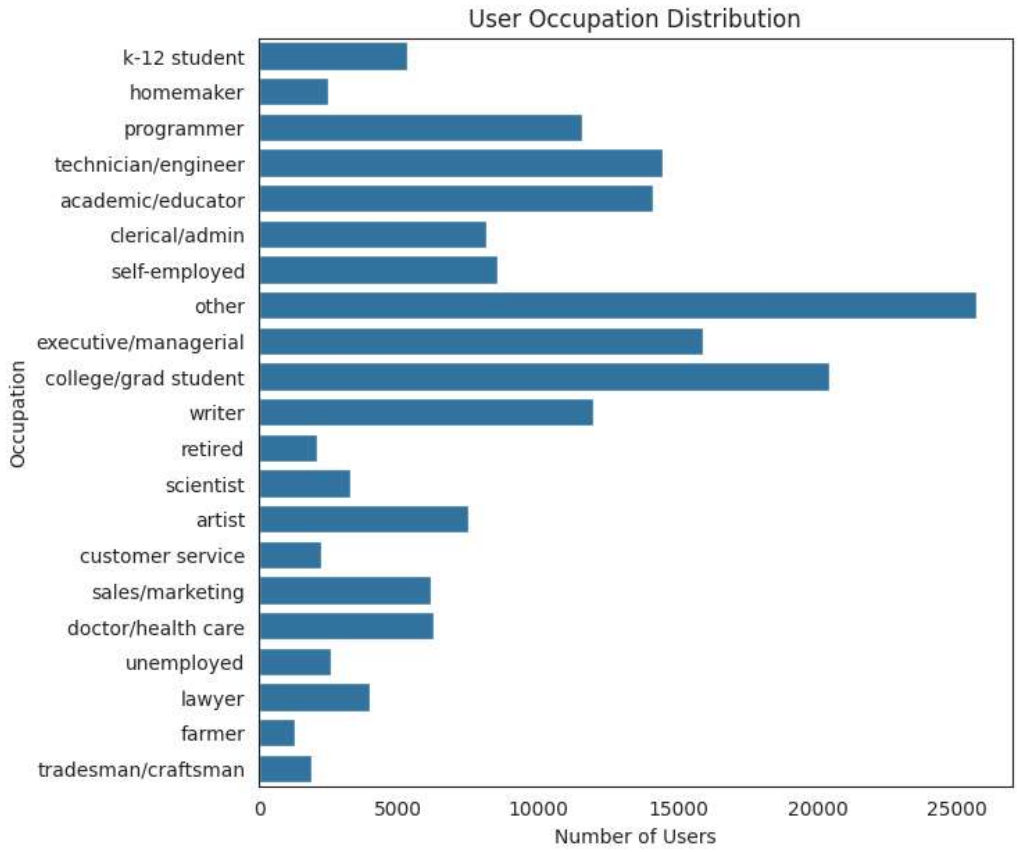
```
x = data['Gender'].value_counts().values
plt.figure(figsize=(7, 6))
plt.pie(x, center=(0, 0), radius=1.5, labels=['M','F'], autopct='%1.1f%%', pctdistance=0.5)
plt.title('User Gender Distribution')
plt.axis('equal')
plt.show()
data['Gender'].value_counts()
```




```
M    134060
F     41761
Name: Gender, dtype: int64
```

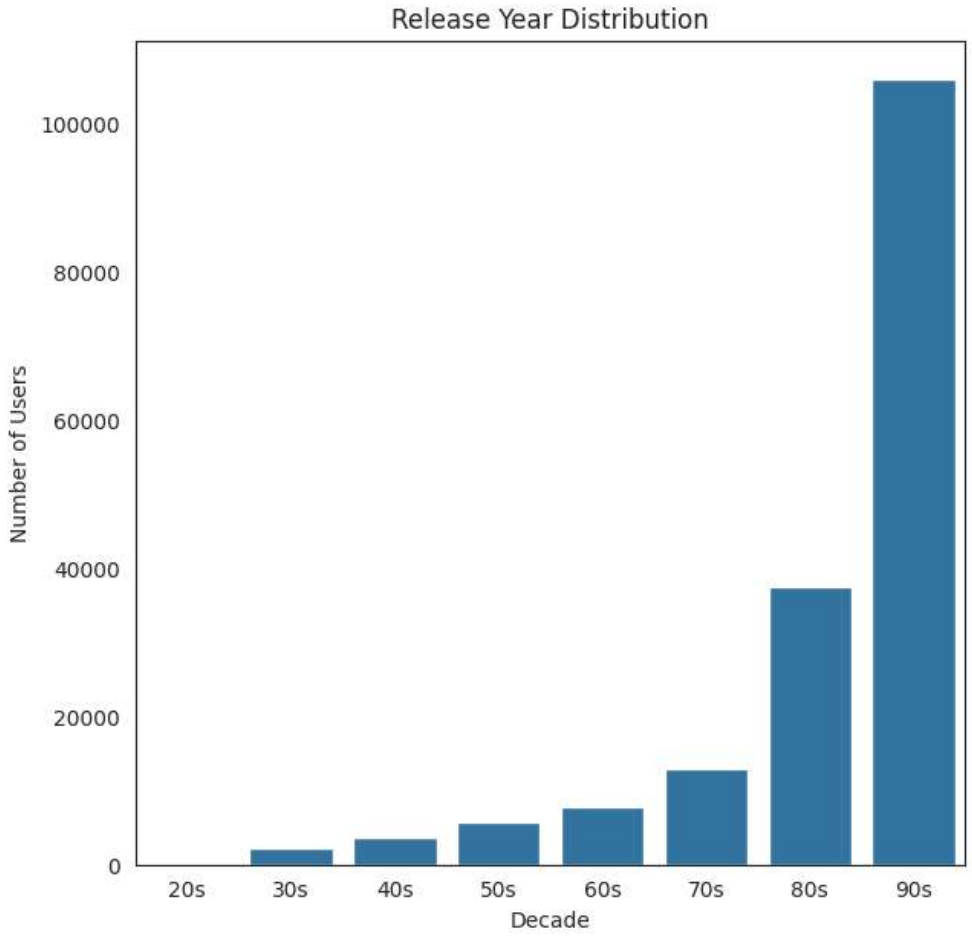
Distribution by Occupation -

```
plt.figure(figsize=(7, 7))
sns.countplot(y='Occupation', data=data)
plt.title('User Occupation Distribution')
plt.xlabel('Number of Users')
plt.ylabel('Occupation')
plt.show()
```



Distribution by Release Year -

```
plt.figure(figsize=(7, 7))
sns.countplot(x='ReleaseDec', data=data)
plt.title('Release Year Distribution')
plt.xlabel('Decade')
plt.ylabel('Number of Users')
plt.show()
```



Grouping the data -

Average rating -

```
data.groupby('Title')['Rating'].mean().sort_values(ascending=False).head(10)
```

```
Title
Eighth Day, The (Le Huitième jour )      5.00
Whatever Happened to Aunt Alice?         5.00
Curdled                                  5.00
I Stand Alone (Seul contre tous)         5.00
Hour of the Pig, The                     5.00
Criminal Lovers (Les Amants Criminels)   5.00
Dancemaker                               5.00
Gate of Heavenly Peace, The              5.00
Autumn Sonata (Höstsonaten )             5.00
Secret Agent                             5.00
Name: Rating, dtype: float64
```

No. of ratings -

```
data.groupby('Title')['Rating'].count().sort_values(ascending=False).head(10)
```

```
Title
American Beauty                                 659
Star Wars: Episode V - The Empire Strikes Back  549
Star Wars: Episode IV - A New Hope              539
Jurassic Park                                   538
Star Wars: Episode VI - Return of the Jedi      537
```

```
Saving Private Ryan                    505
Matrix, The                            491
Terminator 2: Judgment Day             484
Men in Black                           480
Silence of the Lambs, The              473
Name: Rating, dtype: int64
```

```
df = pd.DataFrame(data.groupby('Title')['Rating'].agg([('Avg rating', 'mean')]))
df['No. of ratings'] = pd.DataFrame(data.groupby('Title')['Rating'].count())
```

```
df.sample(10)
```

Title	Avg rating	No. of ratings
Kronos	3.10	20
Raise the Titanic	2.89	9
Carnal Knowledge	3.60	20
Bird on a Wire	2.54	70
My Crazy Life (Mi vida loca)	4.12	8
Outrageous Fortune	2.69	32
Jerry Springer: Ringmaster	1.62	13
Big One, The	4.00	20
Iron Eagle	2.51	37
House on Haunted Hill, The	2.33	63

In our case, we'll be working on a Collaborative Filtering Recommender System.

Collaborative filtering methods are classified as *memory-based* and *model-based*.

Also there are two approaches to this method. A *user-based* approach and an *item-based* approach.

Pivot Table -

Creating a pivot table of movie titles and userid -

```
mat = pd.pivot_table(data, index='UserID', columns='Title', values='Rating', aggfunc='mean')
mat.head(10)
```

Title	$1,000,000 Duck	'Night Mother	'Til There Was You	'burbs, The	...And Justice for All	10 Things I Hate About You	101 Dalmatians	12 Angry Men	13th Warrior, The
UserID									
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
10	NaN	NaN	NaN	4.00	NaN	NaN	NaN	3.00	4.00
100	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1000	NaN	NaN	NaN	NaN	NaN	NaN	4.00	NaN	NaN
1001	NaN	NaN	NaN	NaN	NaN	NaN	3.00	NaN	NaN
1002	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1003	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1004	NaN	NaN	NaN	NaN	NaN	NaN	4.00	NaN	NaN
1005	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1006	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Imputing 'NaN' values with Zero rating -

```
mat.fillna(0, inplace=True)
```

```
mat.shape
```

```
(1114, 3374)
```

Pearson Correlation -

Correlation is a measure that tells how closely two variables move in the same or opposite direction. A positive value indicates that they move in the same direction (i.e. if one increases other increases), where as a negative value indicates the opposite.

The most popular correlation measure for numerical data is Pearson's Correlation. This measures the degree of linear relationship between two numeric variables and lies between -1 to +1. It is represented by 'r'.

- r=1 means perfect positive correlation
- r=-1 means perfect negative correlation
- r=0 means no linear correlation (note, it does not mean no correlation)

Item-based approach:

We will take a movie name as an input from the user and see which other 5 (five) movies have maximum correlation with it.

```
movie_name = input("Enter a movie name: ")
movie_rating = mat[movie_name]
```

```
Enter a movie name: Liar Liar
```

```
similar_movies = mat.corrwith(movie_rating)
```

```
sim_df = pd.DataFrame(similar_movies, columns=['Correlation'])
sim_df.sort_values('Correlation', ascending=False, inplace=True)
```

```
sim_df.iloc[1: , :].head()
```

Title	Correlation
Mrs. Doubtfire	0.53
Dumb & Dumber	0.49
Ace Ventura: Pet Detective	0.46
Wedding Singer, The	0.45
Mask, The	0.45

Cosine Similarity -

[ ] ↳ *7 cells hidden*

Nearest Neighbors -

```
csr_mat = sparse.csr_matrix(mat.T.values)
csr_mat
```

```
<3374x1114 sparse matrix of type '<class 'numpy.float64'>'
        with 175285 stored elements in Compressed Sparse Row format>
```

A sparse matrix or sparse array is a matrix in which most of the elements are zero. A compressed sparse row (CSR) matrix 'M' is represented by three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices. The CSR format stores a sparse m × n matrix M in row form using three (one-dimensional) arrays (V, COL_INDEX, ROW_INDEX).

For example -

Dense matrix representation:

[[1 0 0 0 0 0]

[0 0 2 0 0 1]

[0 0 0 2 0 0]]

Sparse 'row' matrix:

(0, 0) 1

(1, 2) 2

(1, 5) 1

(2, 3) 2

Fitting the model with 'cosine similarity' as the distance metric and 5 (five) as the no. of nearest neighbors.

```
knn = NearestNeighbors(n_neighbors=5, metric='cosine', n_jobs=-1)
knn.fit(csr_mat)
```

```
NearestNeighbors(metric='cosine', n_jobs=-1)
```

Let's make recommendations for a movie of the user's choice -

```
movie_name = input("Enter a movie name: ")
movie_index = mat.columns.get_loc(movie_name)
```

```
Enter a movie name: Liar Liar
```

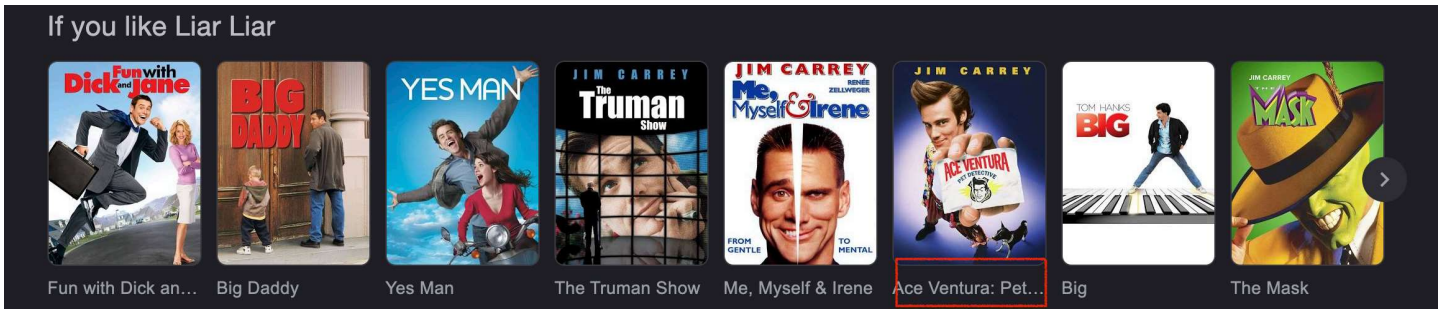
```
distances, indices = knn.kneighbors(mat[movie_name].values.reshape(1, -1), n_neighbors = 11)
```

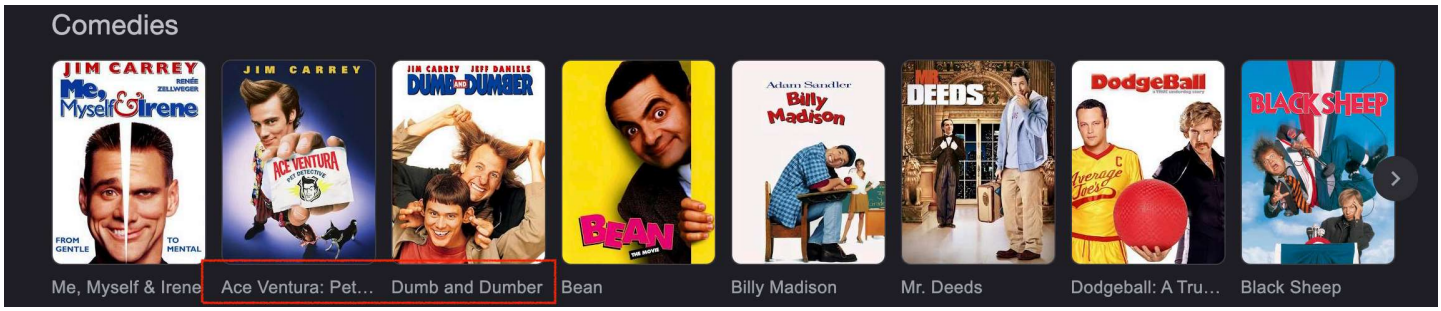
```
for i in range(0, len(distances.flatten())):
    if i == 0:
        print('Recommendations for the movie: {0}\n'.format(movie_name))
    else:
        print('{0}: {1}, with distance of {2}'.format(i, mat.columns[indices.flatten()[i]], round(distances.flatten()[i],
```

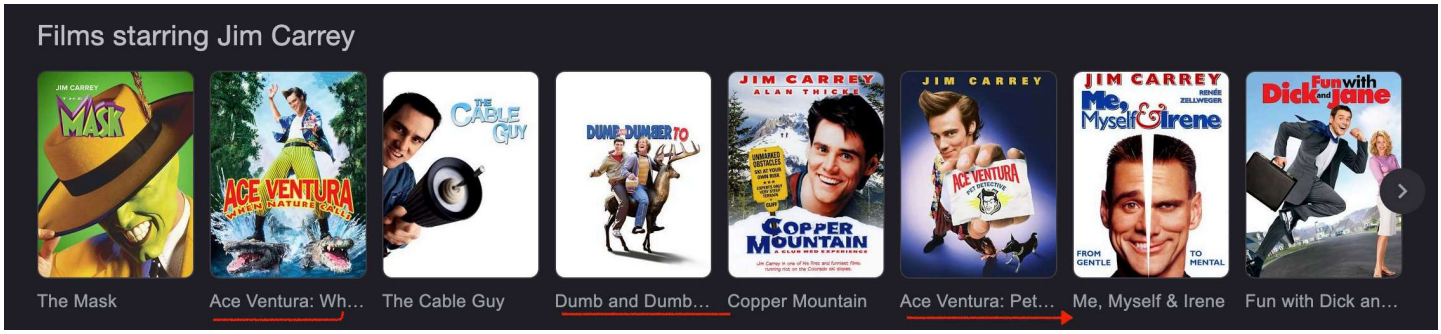
```
Recommendations for the movie: Liar Liar

1: Mrs. Doubtfire, with distance of 0.417
2: Dumb & Dumber, with distance of 0.458
3: Mask, The, with distance of 0.477
4: Ace Ventura: Pet Detective, with distance of 0.481
5: Wedding Singer, The, with distance of 0.482
6: Wayne's World, with distance of 0.485
7: League of Their Own, A, with distance of 0.49
8: Austin Powers: International Man of Mystery, with distance of 0.495
9: Home Alone, with distance of 0.501
10: My Cousin Vinny, with distance of 0.501
```

Let's compare our result with google recomm:







Matrix Factorization -

First we need to create embeddings for both the user as well as the item or movie. For this We have used the Embedding layer from keras.

Creating embeddings for both users and movies -

```
users = data.UserID.unique()
movies = data.MovieID.unique()

userid2idx = {o:i for i,o in enumerate(users)}
movieid2idx = {o:i for i,o in enumerate(movies)}
```

The number of dimensions (Latent Factors) in the embeddings is a hyperparameter to deal with in this implementation of Collaborative Filtering.

```
data['UserID'] = data['UserID'].apply(lambda x: userid2idx[x])
data['MovieID'] = data['MovieID'].apply(lambda x: movieid2idx[x])
split = np.random.rand(len(data)) < 0.8
train = data[split]
valid = data[~split]
print(train.shape, valid.shape)
```

```
(140686, 13) (35135, 13)
```

```
n_movies = len(data['MovieID'].unique())
n_users = len(data['UserID'].unique())

n_latent_factors = 64  # Hyperparamter
```

Specify the input expected to be embedded (Both in user and item embedding). Then use a Embedding layer which expects the number of latent factors in the resulting embedding and also the number of users or items.

```
user_input = Input(shape=(1, ), name='user_input', dtype='int64')

user_embedding = Embedding(n_users, n_latent_factors, name='user_embedding')(user_input)

user_vec = Flatten(name='FlattenUsers')(user_embedding)

movie_input = Input(shape=(1, ), name='movie_input', dtype='int64')
movie_embedding = Embedding(n_movies, n_latent_factors, name='movie_embedding')(movie_input)
movie_vec = Flatten(name='FlattenMovies')(movie_embedding)
```

Then we take the 'Dot-Product' of both the embeddings using the 'merge' layer. Note that 'dot-product' is just a measure of simalrity and we can use any other mode like 'mulitply' or 'cosine simalarity' or 'concatenate' etc.

```
sim = dot([user_vec, movie_vec], name='Simalarity-Dot-Product', axes=1)
model = keras.models.Model([user_input, movie_input], sim)
```

Lastly we make a Keras model from the specified details.

```
model.compile(optimizer=Adam(learning_rate=1e-4), loss='mse')
```

Let's see the model's summary -

```
model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 user_input (InputLayer)     [(None, 1)]                  0         []

 movie_input (InputLayer)    [(None, 1)]                  0         []

 user_embedding (Embedding)  (None, 1, 64)                71296     ['user_input[0][0]']

 movie_embedding (Embedding  (None, 1, 64)                218560    ['movie_input[0][0]']
 )

 FlattenUsers (Flatten)      (None, 64)                   0         ['user_embedding[0][0]']

 FlattenMovies (Flatten)     (None, 64)                   0         ['movie_embedding[0][0]']

 Simalarity-Dot-Product (Do  (None, 1)                    0         ['FlattenUsers[0][0]',
 t)                                                                  'FlattenMovies[0][0]']

==================================================================================================
Total params: 289856 (1.11 MB)
Trainable params: 289856 (1.11 MB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

Note that the metrics used is 'Mean squared Error'. Our aim is to minimize the mse on the training set i.e. over the values which the user has rated.

Model Training -

```
model_hist = model.fit([train.UserID, train.MovieID], train.Rating,
                       batch_size=128, epochs=20,
                       validation_data = ([valid.UserID, valid.MovieID], valid.Rating),
                       verbose=1)
```

```
Epoch 1/20
1100/1100 [==============================] - 12s 9ms/step - loss: 14.0226 - val_loss: 14.0667
Epoch 2/20
1100/1100 [==============================] - 4s 4ms/step - loss: 13.9963 - val_loss: 14.0177
Epoch 3/20
1100/1100 [==============================] - 5s 4ms/step - loss: 13.8111 - val_loss: 13.6245
Epoch 4/20
1100/1100 [==============================] - 4s 4ms/step - loss: 12.9583 - val_loss: 12.2357
Epoch 5/20
1100/1100 [==============================] - 4s 3ms/step - loss: 10.9946 - val_loss: 9.8068
Epoch 6/20
1100/1100 [==============================] - 4s 4ms/step - loss: 8.3516 - val_loss: 7.1229
```

```
Epoch 7/20
1100/1100 [==============================] - 5s 4ms/step - loss: 5.8692 - val_loss: 4.9431
Epoch 8/20
1100/1100 [==============================] - 4s 4ms/step - loss: 4.0862 - val_loss: 3.5485
Epoch 9/20
1100/1100 [==============================] - 4s 4ms/step - loss: 3.0156 - val_loss: 2.7333
Epoch 10/20
1100/1100 [==============================] - 4s 4ms/step - loss: 2.3719 - val_loss: 2.2161
Epoch 11/20
1100/1100 [==============================] - 4s 3ms/step - loss: 1.9469 - val_loss: 1.8616
Epoch 12/20
1100/1100 [==============================] - 5s 4ms/step - loss: 1.6505 - val_loss: 1.6113
Epoch 13/20
1100/1100 [==============================] - 4s 3ms/step - loss: 1.4392 - val_loss: 1.4319
Epoch 14/20
1100/1100 [==============================] - 4s 3ms/step - loss: 1.2866 - val_loss: 1.3017
Epoch 15/20
1100/1100 [==============================] - 4s 4ms/step - loss: 1.1748 - val_loss: 1.2056
Epoch 16/20
1100/1100 [==============================] - 4s 3ms/step - loss: 1.0920 - val_loss: 1.1341
Epoch 17/20
1100/1100 [==============================] - 4s 3ms/step - loss: 1.0297 - val_loss: 1.0798
Epoch 18/20
1100/1100 [==============================] - 4s 4ms/step - loss: 0.9824 - val_loss: 1.0386
Epoch 19/20
1100/1100 [==============================] - 3s 3ms/step - loss: 0.9460 - val_loss: 1.0062
Epoch 20/20
1100/1100 [==============================] - 4s 3ms/step - loss: 0.9175 - val_loss: 0.9808
```

Model Evaluation -

```
y_pred = model.predict([valid.UserID, valid.MovieID], verbose=0)
y_pred_class = np.argmax(y_pred, axis=-1)
```

Calculating the RMSE -

```
from sklearn.metrics import mean_squared_error
rmse = mean_squared_error(valid.Rating, y_pred, squared=False)
print('Root Mean Squared Error: {:.3f}'.format(rmse))
```

```
Root Mean Squared Error: 0.990
```

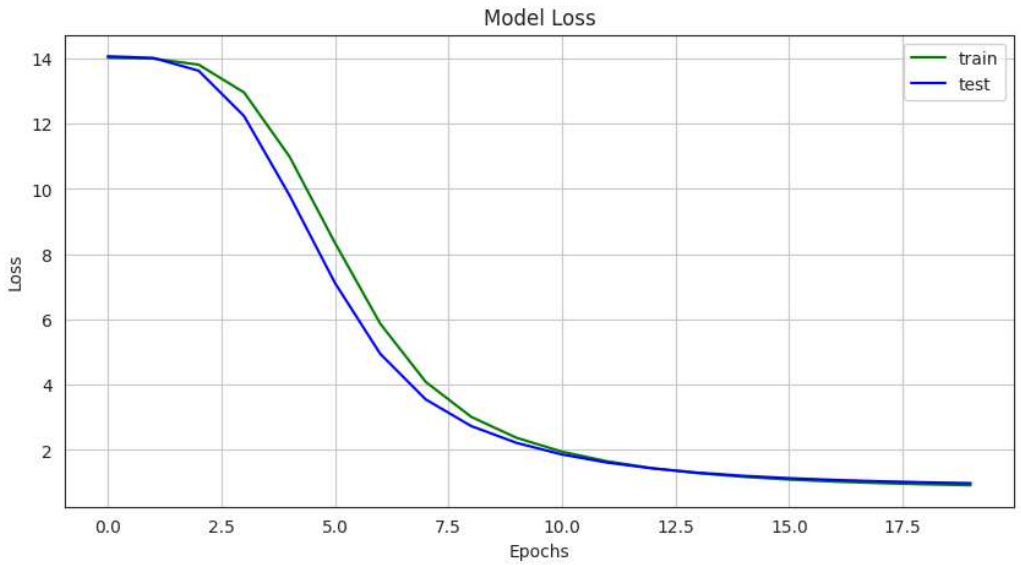
Calculating the MAPE -

```
from sklearn.metrics import mean_absolute_percentage_error
mape =  mean_absolute_percentage_error(valid.Rating, y_pred)
print('Mean Absolute Percentage Error: {:.3f}'.format(mape))
```

```
Mean Absolute Percentage Error: 0.293
```

Plotting the Model Loss -

```
rcParams['figure.figsize'] = 10, 5
plt.plot(model_hist.history['loss'] , 'g')
plt.plot(model_hist.history['val_loss'] , 'b')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(['train', 'test'], loc='upper right')
plt.grid(True)
plt.show()
```



User-based approach: (Optional)

We've given our user 4 (four) movie names and asked them to rate these movies according to their liking.

```
mov_name = ['Mrs. Doubtfire', 'Dumb & Dumber', 'Ace Ventura: Pet Detective', 'Home Alone']
```

```
mov_id = []
for mov in mov_name:
    id = data[data['Title'] == mov]['MovieID'].iloc[0]
    mov_id.append(id)
```

```
mov_rating = list(map(int, input("Rate these movies respectively: ").split()))
```

```
Rate these movies respectively: 3 4 5 5
```

Creating a dataframe for a new user's choices.

```
user_choices = pd.DataFrame({'MovieID': mov_id,
                             'Title': mov_name,
                             'Rating': mov_rating})
user_choices.sort_values(by='MovieID')
```

	MovieID	Title	Rating
2	316	Ace Ventura: Pet Detective	5
3	328	Home Alone	5
1	978	Dumb & Dumber	4
0	1009	Mrs. Doubtfire	3

Users who have watched the same movies as the new users.

```
other_users = data[data['MovieID'].isin(user_choices['MovieID'].values)]
other_users = other_users[['UserID', 'MovieID', 'Rating']]
other_users['UserID'].nunique()
```

```
263
```

Sorting old users by the count of most movies in common with the new user.

```
common_movies = other_users.groupby(['UserID'])
common_movies = sorted(common_movies, key=lambda x: len(x[1]), reverse=True)
common_movies[0]
```

```
(9,
       UserID  MovieID  Rating
 1687       9      978       1
 1705       9      316       1
 1738       9     1009       3
 1749       9      328       2)
```