1 Recommendation System

1.1 Summary

Using a dataset from the MovieLens research lab at the University of Minnesota that consists of about 100,000 user ratings, a recommendation system was created to suggest movies to old and new users that they are likely to enjoy. Two main types of filtering were used to achieve the results of this project; those being collaborative filtering and content based filtering. The available datasets were previewed to see which categories were necessary, which included movie ID, title, genre, user ID, and rating. In order to suggest movies to a user, we need to know which movies the user already saw and what they (as well as others) thought about it. Several models were created and libraries were used to get the optimum results. Those include the Surprise and ALS library to perform collaborative filtering. To achieve the results for content based filtering, a memory based method that is neighborhood based was performed. In the end using the model with the best root mean squared error (KNNBaseline) this project achieved a result of RMSE: 0.5536. In short, the predictions made in this project will be about 0.55 off on a scale from 0 to 5 where this range was the rating for a movie.

1.2 Import necessary libraries

In [1]:

```
# Import necessary libraries
# from pyspark.sql import SparkSession
from surprise import Dataset
from surprise import Reader
from surprise import SVD
from surprise.model_selection import GridSearchCV
from surprise.model_selection import train_test_split
from surprise.model_selection import cross_validate
from surprise.prediction_algorithms import knns
from surprise.similarities import cosine, msd, pearson
from surprise import accuracy
import pandas as pd
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse.linalg import svds
from pyspark.sql import SparkSession
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
import difflib
from re import search
```

In [2]:

```
movies = pd.read_csv('data/movies.csv')
ratings = pd.read_csv('data/ratings.csv')

```

1.3 Learning more about the datasets

Here is where I learned what type of categories each dataset contained.

In [3]:
```
movies.head()
```

Out[3]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

In [4]:
```
ratings.head()
```

Out[4]:

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931

Viewing the shape of each dataset in order to get an idea how many rows and columns I would be dealing with once they were merged.

In [5]:
```
movies.shape
```

Out[5]: (9742, 3)

In [6]:
```
ratings.shape
```

Out[6]: (100836, 4)

In [7]:
```
df = pd.merge(movies, ratings, on='movieId')
df.head()
```

Out[7]:

	movieId	title	genres	userId	rating	timestamp
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	1	4.0	964982703
1	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	5	4.0	847434962
2	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	7	4.5	1106635946
3	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	15	2.5	1510577970
4	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	17	4.5	1305696483

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100836 entries, 0 to 100835
Data columns (total 6 columns):
 #   Column     Non-Null Count   Dtype
---  ------     --------------   -----
 0   movieId    100836 non-null  int64
 1   title      100836 non-null  object
 2   genres     100836 non-null  object
 3   userId     100836 non-null  int64
 4   rating     100836 non-null  float64
 5   timestamp  100836 non-null  int64
dtypes: float64(1), int64(3), object(2)
memory usage: 5.4+ MB
```

To be safe, checking that there were no NaN values.

In [9]:
```
df.isna().sum()
```

Out[9]:
```
movieId      0
title        0
genres       0
userId       0
rating       0
timestamp    0
dtype: int64
```

1.4 Creating df's that will be useful later on

In [10]:
```
df.drop('timestamp', axis=1, inplace= True)
```

In [11]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100836 entries, 0 to 100835
Data columns (total 5 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   movieId  100836 non-null  int64
 1   title    100836 non-null  object
 2   genres   100836 non-null  object
 3   userId   100836 non-null  int64
 4   rating   100836 non-null  float64
dtypes: float64(1), int64(2), object(2)
memory usage: 4.6+ MB
```

Sorting by movie ID here for mostly preference and organization of the dataset.

In [12]:
```
# Sorting by first movie
df.sort_values(by='movieId', ascending=True)
df.head(10)
```

Out[12]:

	movieId	title	genres	userId	rating
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	1	4.0
1	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	5	4.0
2	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	7	4.5
3	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	15	2.5
4	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	17	4.5
5	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	18	3.5
6	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	19	4.0
7	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	21	3.5
8	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	27	3.0
9	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	31	5.0

In [13]:
```
df.shape
```

Out[13]:
```
(100836, 5)
```

1.4.1 Creating main dataset

In [14]:
```
# Separating Genres and making them into list
genre = df['genres'].map(lambda x: x.split('|'))
df['genre'] = genre
df.head(10)
```

Out[14]:

	movieId	title	genres	userId	rating	genre
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	1	4.0	[Adventure, Animation, Children, Comedy, Fantasy]
1	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	5	4.0	[Adventure, Animation, Children, Comedy, Fantasy]
2	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	7	4.5	[Adventure, Animation, Children, Comedy, Fantasy]
3	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	15	2.5	[Adventure, Animation, Children, Comedy, Fantasy]
4	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	17	4.5	[Adventure, Animation, Children, Comedy, Fantasy]
5	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	18	3.5	[Adventure, Animation, Children, Comedy, Fantasy]
6	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	19	4.0	[Adventure, Animation, Children, Comedy, Fantasy]
7	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	21	3.5	[Adventure, Animation, Children, Comedy, Fantasy]
8	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	27	3.0	[Adventure, Animation, Children, Comedy, Fantasy]
9	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	31	5.0	[Adventure, Animation, Children, Comedy, Fantasy]

Turning the genres into a list so that it able to be manipulated easier if need be.

In [15]:
```
df.drop('genres', axis=1, inplace= True)
df.head()
```

Out[15]:

	movieId	title	userId	rating	genre
0	1	Toy Story (1995)	1	4.0	[Adventure, Animation, Children, Comedy, Fantasy]
1	1	Toy Story (1995)	5	4.0	[Adventure, Animation, Children, Comedy, Fantasy]
2	1	Toy Story (1995)	7	4.5	[Adventure, Animation, Children, Comedy, Fantasy]
3	1	Toy Story (1995)	15	2.5	[Adventure, Animation, Children, Comedy, Fantasy]
4	1	Toy Story (1995)	17	4.5	[Adventure, Animation, Children, Comedy, Fantasy]

1.4.2 Creating Ratings Columns

Creating another data frame here to get more information about the ratings.

In [16]:
```
average_ratings = pd.DataFrame(df.groupby('title')['rating'].mean())
average_ratings.sort_values(by='rating', ascending=False)

```

Out[16]:

	rating
title	
Gena the Crocodile (1969)	5.0
True Stories (1986)	5.0
Cosmic Scrat-tastrophe (2015)	5.0
Love and Pigeons (1985)	5.0
Red Sorghum (Hong gao liang) (1987)	5.0
...	...
Don't Look Now (1973)	0.5
Journey 2: The Mysterious Island (2012)	0.5
Joe Dirt 2: Beautiful Loser (2015)	0.5
Jesus Christ Vampire Hunter (2001)	0.5
Fullmetal Alchemist 2018 (2017)	0.5

9719 rows × 1 columns

By sorting by highest rated, I could have an idea of what the best movies looked like, but by looking at this I could tell something was off. Some of these movies could have just a few people who gave it a 5.0. I needed to know the just how many people voted for these movies at this point.

In [17]:
```
average_ratings['Total Ratings'] = pd.DataFrame(df.groupby('title')['rating'].count())

average_ratings.shape

```

Out[17]: (9719, 2)

Checking here for the total amount of ratings placed on all the movies.

In [18]:
```
# Mean rating of all movie averages
mean_rating = average_ratings['rating'].mean()
print(mean_rating)
```

```
3.2623883953257353
```

In [19]:
```
average_ratings.head()
```

Out[19]:

title	rating	Total Ratings
'71 (2014)	4.0	1
'Hellboy': The Seeds of Creation (2004)	4.0	1
'Round Midnight (1986)	3.5	2
'Salem's Lot (2004)	5.0	1
'Til There Was You (1997)	4.0	2

In [21]:
```
# Most rated movie
most_rated = average_ratings.sort_values(by='Total Ratings', ascending=False)
most_rated.head(10)
```

Out[21]:

title	rating	Total Ratings
Forrest Gump (1994)	4.164134	329
Shawshank Redemption, The (1994)	4.429022	317
Pulp Fiction (1994)	4.197068	307
Silence of the Lambs, The (1991)	4.161290	279
Matrix, The (1999)	4.192446	278
Star Wars: Episode IV - A New Hope (1977)	4.231076	251
Jurassic Park (1993)	3.750000	238
Braveheart (1995)	4.031646	237
Terminator 2: Judgment Day (1991)	3.970982	224
Schindler's List (1993)	4.225000	220

After finding the mean of each movie's rating as well as each movie's total rating, the data frame looked better. Sorting now by total ratings, these movies looked more familiar and are known as some of the best.

In [22]:
```
# Number of total ratings
most_rated['Total Ratings'].sum()
```

Out[22]: 100836

1.4.3 Finding out more about the ratings

In [24]:
```
# Checking to see the number of people who have a certain rating
df.rating.value_counts()
```

Out[24]:
```
4.0    26818
3.0    20047
5.0    13211
3.5    13136
4.5     8551
2.0     7551
2.5     5550
1.0     2811
1.5     1791
0.5     1370
Name: rating, dtype: int64
```

In [25]:
```
# Getting min and max rating that was given
lower_rating = df['rating'].min()
upper_rating = df['rating'].max()
print('User rating range: {0} to {1}'.format(lower_rating, upper_rating))
```

```
User rating range: 0.5 to 5.0
```

Finding the min and max rating for all the movies was important here because this information is needed for the Surprise library calculations.

In [26]:
```
# Create Surprise Dataset
reader = Reader(rating_scale=(0.5,5))
data = Dataset.load_from_df(df[['userId', 'movieId', 'rating']], reader)
```

In [27]:
```
svd = SVD(verbose= True, n_epochs=10)
cross_validate(svd, data, measures= ['RMSE', 'MAE'], cv=3, verbose= True)
```

```
Processing epoch 0
Processing epoch 1
Processing epoch 2
Processing epoch 3
Processing epoch 4
Processing epoch 5
Processing epoch 6
Processing epoch 7
Processing epoch 8
Processing epoch 9
Processing epoch 0
Processing epoch 1
Processing epoch 2
Processing epoch 3
Processing epoch 4
Processing epoch 5
Processing epoch 6
Processing epoch 7
Processing epoch 8
Processing epoch 9
Processing epoch 0
Processing epoch 1
Processing epoch 2
Processing epoch 3
Processing epoch 4
Processing epoch 5
Processing epoch 6
Processing epoch 7
Processing epoch 8
Processing epoch 9
Evaluating RMSE, MAE of algorithm SVD on 3 split(s).

                  Fold 1  Fold 2  Fold 3  Mean    Std
RMSE (testset)    0.8871  0.8849  0.8883  0.8868  0.0014
MAE (testset)     0.6845  0.6832  0.6848  0.6842  0.0007
Fit time          1.73    1.74    1.76    1.74    0.01
Test time         0.17    0.22    0.23    0.21    0.03
```

Out[27]:
```
{'test_rmse': array([0.88713344, 0.88493684, 0.88828583]),
 'test_mae': array([0.68454389, 0.68318737, 0.68478607]),
 'fit_time': (1.7331767082214355, 1.7393770217895508, 1.7623910903930664),
 'test_time': (0.17146825790405273, 0.22491097450256348, 0.22838091850280762)}
```

This was my first go around of using the Surprise library features. Basically a test run for what range the RMSE I should be expecting.

1.4.4 Creating train and test sets

In [28]:
```
# Split into train and test set
trainset, testset = train_test_split(data, test_size=0.2)
```

In [29]:
```
print('Type trainset : ', type(trainset),'\n')
print('Type testset :',type(testset))
```

```
Type trainset :  <class 'surprise.trainset.Trainset'>

Type testset : <class 'list'>
```

In [30]:
```
# Checking how big the test set is as well as what info is included in it
print(len(testset))
print(testset[0])
```

```
20168
(414, 30707, 4.5)
```

In [31]:
```
print('Number of users: ', trainset.n_users, '\n')
print('Number of items: ', trainset.n_items, '\n')

```

```
Number of users:  610

Number of items:  8935
```

Getting an idea here of whether I should set user_based to True or False. The higher number would result in more computing time.

In [32]:
```
# Because of fewer users than items, it is more efficient to calculate
# user-user similarity rather than item-item
# However since this is set to false, this is using item-item
sim_cos = {'name':'cosine', 'user_based':False}
```

In [33]:
```
basic = knns.KNNBasic(sim_options=sim_cos)
# Fit model
basic.fit(data.build_full_trainset())
```

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

Out[33]:
```
<surprise.prediction_algorithms.knns.KNNBasic at 0x7f9071303040>
```

In [34]:
```
# Similarity metrics of each of the movie to one another
basic.sim
```

Out[34]:
```
array([[1.        , 0.96446411, 0.97154149, ..., 0.        , 0.        ,
        0.        ],
       [0.96446411, 1.        , 0.93890126, ..., 0.        , 0.        ,
        0.        ],
       [0.97154149, 0.93890126, 1.        , ..., 0.        , 0.        ,
        0.        ],
       ...,
       [0.        , 0.        , 0.        , ..., 1.        , 1.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 1.        , 1.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        1.        ]])
```

This was the first similarity metric produced, but this didn't look like much to me so I thought I would understand it better by putting it in a data frame.

In [35]:
```
# Verifing size of dataset
basic.sim.shape
```

Out[35]:
```
(9724, 9724)
```

In [36]:
```
# getting better idea of what array of similarity metrics looks like
basic_df = pd.DataFrame(basic.sim)
basic_df.head()
```

Out[36]:

	0	1	2	3	4	5	6	7	8	9	...	9714	9715	9
0	1.000000	0.964464	0.971541	0.983870	0.955044	0.966239	0.944239	0.988721	0.950356	0.952135	...	0.0	0.0	
1	0.964464	1.000000	0.938901	0.960988	0.968546	0.954224	0.968611	0.969442	0.943370	0.946736	...	0.0	0.0	
2	0.971541	0.938901	1.000000	1.000000	0.972298	0.933685	0.971202	0.973985	0.942695	0.921344	...	0.0	0.0	
3	0.983870	0.960988	1.000000	1.000000	0.972003	1.000000	0.943293	1.000000	0.000000	0.973708	...	0.0	0.0	
4	0.955044	0.968546	0.972298	0.972003	1.000000	0.966924	0.974637	0.960928	0.963287	0.952833	...	0.0	0.0	

5 rows × 9724 columns

This helped me understand that it was comparing each movie to one another and producing a number of how similar that movie was to the other. The highest number meaning it was very similar.

1.4.5 Finding the model that will give the lowest RMSE

In [37]:
```
# Test model to determine how well it performed
predictions = basic.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 0.9031
0.9030818565366927
```

Results mean model is off by about 0.9013 points

In [38]:
```
# Pearson Correlation
sim_pearson = {'name':'pearson', 'user_based':False}
basic_pearson = knns.KNNBasic(sim_options=sim_pearson)
basic_pearson.fit(data.build_full_trainset())
predictions = basic_pearson.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
RMSE: 0.6960
0.6959736433654313
```

In [39]:
```
# KNN with Means. Same as basic KNN model but takes into account the mean rating
# of each user/item
sim_pearson = {'name':'pearson', 'user_based':True}
knn_means = knns.KNNWithMeans(sim_options=sim_pearson)
knn_means.fit(data.build_full_trainset())
predictions = knn_means.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
RMSE: 0.6053
0.6053176017550091
```

In [40]:
```
# KNNBaseline method - This adds in bias term that is calculated by way of
# minimizing a cost function
sim_pearson = {'name':'pearson', 'user_based':False}
knn_baseline = knns.KNNBaseline(sim_options=sim_pearson)
knn_baseline.fit(data.build_full_trainset())
predictions = knn_baseline.test(testset)
print(accuracy.rmse(predictions))
```

```
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
RMSE: 0.5536
0.5536483298449599
```

KNNBaseline method had the best results with 0.5941 RMSE (user_based set to True). And RMSE: 0.5531 when user_based set to False.

In [41]:
```
# Storing knn metrics in a variable
knn_metrics = knn_baseline.sim
```

In [42]:
```
knn_bsl_df = pd.DataFrame(knn_baseline.sim)
knn_bsl_df.head()
```

Out[42]:

	0	1	2	3	4	5	6	7	8	9	...	9714	9715
0	1.000000	0.330978	0.487109	1.000000	0.310971	0.106465	0.208402	0.968246	0.095913	-0.021409	...	0.0	0.0
1	0.330978	1.000000	0.419564	0.000000	0.562791	0.163510	0.430261	0.415227	0.277350	0.016626	...	0.0	0.0
2	0.487109	0.419564	1.000000	0.000000	0.602266	0.345069	0.554088	0.333333	0.458591	-0.050276	...	0.0	0.0
3	1.000000	0.000000	0.000000	1.000000	0.654654	0.000000	0.203653	0.000000	0.000000	0.870388	...	0.0	0.0
4	0.310971	0.562791	0.602266	0.654654	1.000000	0.291302	0.609119	0.555556	0.319173	0.218263	...	0.0	0.0

5 rows × 9724 columns

In [43]:
```
# Checking a specific movie's metrics by their movie ID
knn_bsl_df.loc[knn_bsl_df.index == 314]
```

Out[43]:

	0	1	2	3	4	5	6	7	8	9	...	9714	9715	9716
314	0.303465	0.367247	0.534682	0.388514	0.349541	0.137421	0.106567	0.65602	0.0	0.217441	...	0.0	0.0	0.0

1 rows × 9724 columns

In [44]:
```
# Matrix Factorization
svd = SVD(n_factors=100, n_epochs=10, lr_all=0.005, reg_all=0.4)
#svd.fit(trainset)
svd.fit(data.build_full_trainset())
predictions = svd.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 0.8577
0.857650177663376
```

The above methods tested were to find the right combination that would produce the lowest RMSE. Turns out that was KNNBaseline.

In [45]:
```
# Make Predictions
user_3_prediction = svd.predict('3', '25')
user_3_prediction
```

Out[45]:
```
Prediction(uid='3', iid='25', r_ui=None, est=3.501556983616962, details={'was_impossible':
False})
```

Here I was getting familiar with how the prediction feature worked to apply later on.

In [46]:
```
# Retrieve movie name
def name_retriever(movie_id):
    for movie in df:
       return df.loc[df['movieId']==movie_id].title.values[0]


```

In [47]:
```
# testing out name retriever function
name_retriever(318)
```

Out[47]: 'Shawshank Redemption, The (1994)'

I determined a function needed to be created to turn movie ID's into their actual movie name to lessen the confusion.

In [48]:
```
# Pandas df of unique movie ID's
unique_iids = df['movieId'].drop_duplicates(keep='first')
```

In [49]:
```
# Number of different movies
unique_iids.shape
```

Out[49]: (9724,)

In [24]:
```
# Number of different users
unique_users = df['userId'].drop_duplicates(keep='first')
unique_users.shape
```

Out[24]: (610,)

A dataset of unique users and movies needed to be created so that it could be compared to later on.

In [50]:
```
# Find unrated movies by a user function
def find_unrated(user, movie_df):
    movId = df.loc[df['userId'] == user, 'movieId']
    # Remove the iids that user n has rated from the list of all movie ids
    user_unrated = pd.concat([unique_iids,movId]).drop_duplicates(keep=False)
    user_unrated = user_unrated.to_frame()
    return user_unrated.head()


```

In [51]:
```
# testig out find_unrated function
find_unrated(4, df)
```

Out[51]:

	movieId
0	1
215	2
325	3
377	4
384	5

This function was very important and necessary to create to solve the first part of this project. I needed to know which movies any user in this dataset did not see. It wouldn't make much sense to try to predict a movie rating that a user already rated.

1.5 Recommending movies for user 4 using SVD

Here is where I put my final idea into practice. Sorting from a list of estimated ratings of movies the user has not seen to produce a top 5 movie recommendation.

In [52]:
```
# Recommending movies for user 3 using SVD
svd = SVD(n_factors=100, n_epochs=10, lr_all=0.005, reg_all=0.4)
svd.fit(data.build_full_trainset())
```

Out[52]: `<surprise.prediction_algorithms.matrix_factorization.SVD at 0x7f8f1f085250>`

In [53]:
```
# Putting unique movie ids in a variable
unique_ids = df['movieId'].unique()

#Get list of unique ids for user 4
newdf = df.loc[df['userId'] == 4 , 'movieId']

# Remove rated movies
movies_to_predict = np.setdiff1d(unique_ids, newdf)

```

In [54]:
```
# Make prediction of unrated movie
# svd.predict(uid='3',iid='5')
```

In [55]:
```
# Creating top 5 recommendations for user 4
recs = []
for iid in movies_to_predict:
    recs.append((iid, svd.predict(uid='4',iid=iid).est))
top_5_rec = pd.DataFrame(recs,columns=['iid','predictions']).sort_values('predictions',

top_5_rec
```

Out[55]:

	iid	predictions
256	318	4.101610
557	750	4.066828
824	1204	4.059426
612	858	4.016887
42	50	4.016261

In [56]:
```
# Applying name retriever function to the movie IDS of recommendations
top_5_rec['Movie'] = top_5_rec['iid'].apply(name_retriever)
top_5_rec
```

Out[56]:

	iid	predictions	Movie
256	318	4.101610	Shawshank Redemption, The (1994)
557	750	4.066828	Dr. Strangelove or: How I Learned to Stop Worr...
824	1204	4.059426	Lawrence of Arabia (1962)
612	858	4.016887	Godfather, The (1972)
42	50	4.016261	Usual Suspects, The (1995)

1.6 Recommending movies for user 4 using KNNBaseline model

Now that I knew the idea could work, I applied the same function but with KNNBaseline this time to get an even better top 5 recommendation for any user.

In [57]:
```
cross_validate(knn_baseline, data, measures= ['RMSE', 'MAE'], cv=3, verbose= True)
```

```
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBaseline on 3 split(s).

                  Fold 1  Fold 2  Fold 3  Mean    Std
RMSE (testset)    0.8824  0.8843  0.8849  0.8839  0.0011
MAE (testset)     0.6777  0.6790  0.6807  0.6791  0.0012
Fit time          13.05   13.15   12.82   13.01   0.14
Test time         8.13    7.86    7.89    7.96    0.12
```

Out[57]:
```
{'test_rmse': array([0.88239329, 0.88429518, 0.88491002]),
 'test_mae': array([0.67769519, 0.67903973, 0.68065938]),
 'fit_time': (13.04737401008606, 13.151367902755737, 12.81882095336914),
 'test_time': (8.130614995956421, 7.863303184509277, 7.889848232269287)}
```

In [58]:
```
# Doing the same thing but with KNNBaseline method
recs = []
for iid in movies_to_predict:
    recs.append((iid, knn_baseline.predict(uid='4',iid=iid).est))
top_5_rec = pd.DataFrame(recs,columns=['iid','predictions']).sort_values('predictions',

top_5_rec
```

Out[58]:

	iid	predictions
256	318	4.403969
612	858	4.328440
824	1204	4.286615
557	750	4.269324
42	50	4.250099

In [59]:
```
# Finding out name of movie recommendations
top_5_rec['Movie'] = top_5_rec['iid'].apply(name_retriever)
top_5_rec
```

Out[59]:

	iid	predictions	Movie
256	318	4.403969	Shawshank Redemption, The (1994)
612	858	4.328440	Godfather, The (1972)
824	1204	4.286615	Lawrence of Arabia (1962)
557	750	4.269324	Dr. Strangelove or: How I Learned to Stop Worr...
42	50	4.250099	Usual Suspects, The (1995)

From this top 5 list, it is slightly different from that of the SVD method.

1.7 Using ALS

For sanity purposes, I also tried out the ALS method just in case it had a better RMSE score.

In [60]:
```
print('Using ALS')
bsl_options = {'method': 'als',
               'n_epochs': 5,
               'reg_u': 12,
               'reg_i': 5
               }
algo = knns.KNNBaseline(bsl_options=bsl_options)
cross_validate(algo, data, measures=['RMSE'], cv=3, verbose=False)
```

```
Using ALS
Estimating biases using als...
Computing the msd similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the msd similarity matrix...
Done computing similarity matrix.
Estimating biases using als...
Computing the msd similarity matrix...
Done computing similarity matrix.
```

Out[60]:
```
{'test_rmse': array([0.87906848, 0.8800158 , 0.88121457]),
 'fit_time': (0.15366196632385254, 0.1632378101348877, 0.16462492942810059),
 'test_time': (1.8678550720214844, 1.837568998336792, 1.9958178997039795)}
```

In [61]:
```
# User train test split to sample a trainset and testset
algo = knns.KNNBaseline(bsl_options=bsl_options)
# Use full dataset
predictions = algo.fit(data.build_full_trainset()).test(testset)

accuracy.rmse(predictions)
```

```
Estimating biases using als...
Computing the msd similarity matrix...
Done computing similarity matrix.
RMSE: 0.6757
```

Out[61]:
```
0.675702953578366
```

RMSE using ALS turned out to be 0.6743 using KNNBaseline.

2 Fix to cold start problem: User input & Content Based Filtering

To solve this problem, I thought to myself what does companies like Netflix and Hulu currently do for their new users? The answer is they ask them what type of genre, show, or movies they like. In this case, I sorted earlier in this project the highest rated movies and gave the user the option to choose from the top ten. Chances are, the user has seen and possibly liked these very popular movies.

In [62]:
```
# Get list of top ten movies rated
top_10_rated = most_rated.head(10).index.tolist()
```

In [63]:
```python
# prompt user to input their favorite movie out of the top 10 rated movies
print('Here are the top 10 most rated movies.\n')
print('Which one of these movies do you like? \n')
print(top_10_rated)

```

```
Here are the top 10 most rated movies.

Which one of these movies do you like?

['Forrest Gump (1994)', 'Shawshank Redemption, The (1994)', 'Pulp Fiction (1994)', 'Silenc
e of the Lambs, The (1991)', 'Matrix, The (1999)', 'Star Wars: Episode IV - A New Hope (19
77)', 'Jurassic Park (1993)', 'Braveheart (1995)', 'Terminator 2: Judgment Day (1991)', "S
chindler's List (1993)"]
```

In [64]:
```python
movie_name = input('Enter your favorite movie : ')
```

```
Enter your favorite movie : Schindler's List (1993)
```

In [65]:
```python
# Create list with all movies in dataset
list_of_titles = df['title'].drop_duplicates().tolist()
#print(list_of_titles)
```

In [66]:
```python
# Get index of movie user likes
#Get movie ID instead
def get_movie_index_from_title(title):
    return df[df['title']==movie_name].index.values[0]
```

In [67]:
```python
movie_index = get_movie_index_from_title(movie_name)
```

In [86]:
```python
# Getting index number from original dataset
old_index = df[df['title']==movie_name].index.values[0]
old_index
```

Out[86]:
```
14106
```

Get the index of the movie the user selected from the main dataset. The problem here is that this index number means nothing when compared to the movie index from the similarity metric.

In [80]:
```python
# resetting index
reset_ind = unique_iids.reset_index()
reset_ind
```

Out[80]:

	index	movieId
0	0	1
1	215	2
2	325	3
3	377	4
4	384	5
...	...	...
9719	100831	193581
9720	100832	193583
9721	100833	193585
9722	100834	193587
9723	100835	193609

9724 rows × 2 columns

Another problem that needed to be solved was the index number was too high. 14106 is way more than the actual unique amount of movies. Thus, each index needed another index that would be the same amount in the similarity metric array.

In [89]:
```
# getting new index from unique movie ids
new_index =reset_ind[reset_ind['index']==old_index].index.values[0]
new_index
```

Out[89]: 461

Now I have an index I can work with that I can link back to the original movie the user choose.

In [88]:
```
# matching old index with new index
df['title'][df.index == old_index]
```

Out[88]:
```
14106    Schindler's List (1993)
Name: title, dtype: object
```

In [72]:
```
# Comparing user input to dataset
df.loc[df['title'] == movie_name]
```

Out[72]:

	movieId	title	userId	rating	genre
14106	527	Schindler's List (1993)	1	5.0	[Drama, War]
14107	527	Schindler's List (1993)	3	0.5	[Drama, War]
14108	527	Schindler's List (1993)	5	5.0	[Drama, War]
14109	527	Schindler's List (1993)	6	3.0	[Drama, War]
14110	527	Schindler's List (1993)	8	5.0	[Drama, War]
...	...	...	...	...	...
14321	527	Schindler's List (1993)	603	3.0	[Drama, War]
14322	527	Schindler's List (1993)	606	5.0	[Drama, War]
14323	527	Schindler's List (1993)	607	5.0	[Drama, War]
14324	527	Schindler's List (1993)	608	4.0	[Drama, War]
14325	527	Schindler's List (1993)	610	3.5	[Drama, War]

220 rows × 5 columns

In [90]:
```
# Generating similar movies matrix
similar_movies = list(enumerate(knn_metrics[new_index]))
```

In [74]:
```
# Sorting movies based on simmularity score
# Sort movies in descending order and get second value of tuple
sorted_similar_movies = sorted(similar_movies, key= lambda x:x[1], reverse = True)
#print(sorted_similar_movies)
```

Using the new index created, this was matched with the same index in the similarity metric array.

In [75]:
```
# Creating function to get movie title from index
def get_title_from_index(index):
    orig_id = reset_ids[reset_ids.index == index]["index"].values[0]
    return df[df.index==orig_id].title.values[0]
```

In [91]:
```
# Testing said function
get_title_from_index(461)
```

Out[91]: "Schindler's List (1993)"

Another function needed to be created now to get the movie title from the new index I have been using.

In [92]:
```
# Print the name of similar movies based on index
print(f'Here are 10 movies we think you may like because you chose {movie_name}. ')
i = 0
for movie in sorted_similar_movies:
    print(get_title_from_index(movie[0]))
    i=i+1
    if i>10:
        break
```

```
Here are 10 movies we think you may like because you chose Schindler's List (1993).
Eye for an Eye (1996)
Amazing Panda Adventure, The (1995)
Blue in the Face (1995)
Party Girl (1995)
Farinelli: il castrato (1994)
Love Affair (1994)
Priest (1994)
Forrest Gump (1994)
Blue Chips (1994)
Lassie (1994)
Puppet Masters, The (1994)
```

To put it all back together, this loop is created to go through all the similar movie (that have been sorted at this point from highest similarity score to lowest) and apply the function created above to turn that new index into a movie title.

3 Conclusion

Through a lot of trial and error, the optimum dataset and models were created to obtain the best result. Information I thought I would need like genre or minimum votes to accept in my model were ultimately removed because it was causing too much clutter in the final results.

A dataset of just user ID, movie ID, rating, and title were all that were needed. Experimenting between which similarity metrics would serve the model better also turned out to be fruitful in terms of the least amount of error. A item-item similarity metrics was ultimately selected to solve the cold start problem despite a slightly longer computing time.

In addition, multiple methods were tested to find the best RMSE. Those included the Surprise library K nearest neighbor algorithms such as KNNBasic, Pearson Correlation, KNNWithMeans, KNNBaseline, and SVD matrix factorization. Furthermore, the Alternate Least Squares (ALS) method was also attempted. Through all these models, the method that showed to lowest RMSE was the KNNBaseline method where it used an item to item similarity metric. The result of this was a RMSE of 0.5536 which was far less than the other RMSE's.

Putting it all together the first part of this project was solved using collaborative filtering. By providing a user ID already in the dataset with a movie ID the user has not seen yet, an estimate can be predicted on what that specific user may rate that movie. Once that is figured out, a simple sort to find the highest estimate a user may give a movie and be returned.

As mentioned earlier, a similarity metric was created to help solve the cold start problem. Content based filter was used in this case, or in other words if a user likes a certain movie, they may like similar movies to that one. The metric created was helpful because it was an array of how alike each movie was to one another. From there, the new user is asked which out of the top rated movies have they seen and liked. In this project case, the user selected Schindler's List and the movies most similar to that one were returned.