In [114]:
```
! pip install scikit-surprise
```
...

Recommendation System Project

Problem Statement

For this project I will be making movie recommendations based on the MovieLens (https://grouplens.org/datasets/movielens/latest/ (https://grouplens.org/datasets/movielens/latest/)) dataset from the GroupLens research lab at the University of Minnesota. I will be using the "small" dataset containing 100,000 user ratings.

My main goal is to build a model that provides top 5 movie recommendations to a user, based on their ratings of other movies. Also, my recommendation system will be using collaborative filtering as the primary mechanism and content-based filtering to address the cold start problem.

Collaborative filtering

To build a system that can automatically recommend items to users based on the preferences of other users,I will need to answer these questions:

1. How to determine which users or items are similar to one another?
2. Given that I know which users are similar, how do I determine the rating that a user would give to an item based on the ratings of similar users?
3. How do I measure the accuracy of the ratings you calculate?

Import libraries

In [21]:
```
# Data manipulation
import pandas as pd
import numpy as np

# Data visualization
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# Building recommender systems
from surprise import Reader, BaselineOnly, KNNBasic
from surprise import SVD
from surprise import Dataset
from surprise.model_selection import cross_validate

# Others
from scipy.stats import pearsonr
from tqdm.auto import tqdm

```

Reading and Exploring the data

In [22]:
```
links = pd.read_csv("ml-latest-small/links.csv")
movies = pd.read_csv("ml-latest-small/movies.csv")
ratings = pd.read_csv("ml-latest-small/ratings.csv")
tags = pd.read_csv("ml-latest-small/tags.csv")
```

In [23]:

```python
# Display the first 5 entries in each dataframe
display(links.head())
display(movies.head())
display(ratings.head())
display(tags.head())
```

	movieId	imdbId	tmdbId
0	1	114709	862.0
1	2	113497	8844.0
2	3	113228	15602.0
3	4	114885	31357.0
4	5	113041	11862.0

	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

	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

	userId	movieId	tag	timestamp
0	2	60756	funny	1445714994
1	2	60756	Highly quotable	1445714996
2	2	60756	will ferrell	1445714992
3	2	89774	Boxing story	1445715207
4	2	89774	MMA	1445715200

In [24]:
```
# Determine and display the min and max ratings received
min_rating = ratings['rating'].min()
max_rating = ratings['rating'].max()
print('Lowest rating: {}'.format(min_rating))
print('Highest rating: {}'.format(max_rating))


```

```
Lowest rating: 0.5
Highest rating: 5.0
```

Movies are rated between 0 and 5 with the lowest rating being 0.5 and the highest 5.

In [46]:
```
ratings.describe()
```

Out[46]:

	userId	movieId	rating	timestamp
count	100836.000000	100836.000000	100836.000000	1.008360e+05
mean	326.127564	19435.295718	3.501557	1.205946e+09
std	182.618491	35530.987199	1.042529	2.162610e+08
min	1.000000	1.000000	0.500000	8.281246e+08
25%	177.000000	1199.000000	3.000000	1.019124e+09
50%	325.000000	2991.000000	3.500000	1.186087e+09
75%	477.000000	8122.000000	4.000000	1.435994e+09
max	610.000000	193609.000000	5.000000	1.537799e+09

The average rating across all users and movies is 3.5.

In [25]:
```
# is any row null in links
links.isnull().any()
```

Out[25]:
```
movieId    False
imdbId     False
tmdbId      True
dtype: bool
```

In [26]:
```
# lets drop null rows
links = links.dropna()
```

In [27]:
```

# is any row has null
movies.isnull().any()
```

Out[27]:
```
movieId    False
title      False
genres     False
dtype: bool
```

In [28]:
```
# check ratings
ratings.isnull().any()
```

Out[28]:
```
userId       False
movieId      False
rating       False
timestamp    False
dtype: bool
```

In [29]:
```
# check for tags
tags.isnull().any()
```

Out[29]:
```
userId       False
movieId      False
tag          False
timestamp    False
dtype: bool
```

Data preprocessing

In [30]:

```python
#  movie: Adventure|Children|Fantasy  --> 1 0 1 1 0 0
movies = pd.read_csv("ml-latest-small/movies.csv")
all_genres = set()

#  define all genres that can be foung in the "genres" column
for movie_genres in movies['genres']:
    all_genres = all_genres | set(movie_genres.split('|'))  # merge two

#  just adding new columns to the dataframe and filling them with zeros
for g in all_genres:
    movies[g] = 0

#  put 1's in right places
for i in tqdm(range(len(movies['genres']))):
    movie_genres = movies['genres'][i]
    for g in all_genres:
        if g in set(movie_genres.split('|')):
            movies[g][i] = 1


all_genres = list(all_genres)
movies = movies.set_index('movieId')
movies.head()
```

```
HBox(children=(IntProgress(value=0, max=9742), HTML(value='')))

/Users/alinakorsunenko/opt/anaconda3/envs/learn-env/lib/python3.6/site-packages/ipykernel_launcher.py:18: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
```

Out[30]:

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

5 rows × 22 columns

Cold start developing

In [31]:
```python
# merging two dataframes "movies.csv" and "ratings.csv"
movie_data = pd.merge(ratings, movies, on = 'movieId')
movie_data.head()
```

Out[31]:

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

5 rows × 26 columns

In [32]:
```python
# find out the average rating for each and every movie in the dataset.
movie_data.groupby('title')['rating'].mean().head()
```

Out[32]:
```
title
'71 (2014)                                 4.0
'Hellboy': The Seeds of Creation (2004)    4.0
'Round Midnight (1986)                     3.5
'Salem's Lot (2004)                        5.0
'Til There Was You (1997)                  4.0
Name: rating, dtype: float64
```

In [33]:
```python
# sort the ratings in the ascending order of their average ratings:
movie_data.groupby('title')['rating'].mean().sort_values(ascending = Fa
```

Out[33]:
```
title
Karlson Returns (1970)                           5.0
Winter in Prostokvashino (1984)                  5.0
My Love (2006)                                   5.0
Sorority House Massacre II (1990)                5.0
Winnie the Pooh and the Day of Concern (1972)    5.0
Name: rating, dtype: float64
```

A movie can make it to the top of the above list even if only a single user has given it five stars. Therefore, the above results can be misleading. Normally, a movie which is really a good one gets a higher rating by a large number of users.

In [34]:
```
# plot the total number of ratings for a movie:
movie_data.groupby('title')['rating'].count().sort_values(ascending = F
```

Out[34]:
```
title
Forrest Gump (1994)                 329
Shawshank Redemption, The (1994)    317
Pulp Fiction (1994)                 307
Silence of the Lambs, The (1991)    279
Matrix, The (1999)                  278
Name: rating, dtype: int64
```

The above list supports the point that good movies normally receive higher ratings.

In [35]:
```
# add the average rating of each movie to ratings_mean_count dataframe
ratings_mean_count = pd.DataFrame(movie_data.groupby('title')[['rating'
```

In [36]:
```
# add the number of ratings for a movie to the ratings_mean_count dataf
ratings_mean_count['rating_counts'] = pd.DataFrame(movie_data.groupby('
```

In [41]:
```
# convert movieId to int
ratings_mean_count['movieId'] = ratings_mean_count['movieId'].astype(in
ratings_mean_count.head()
```

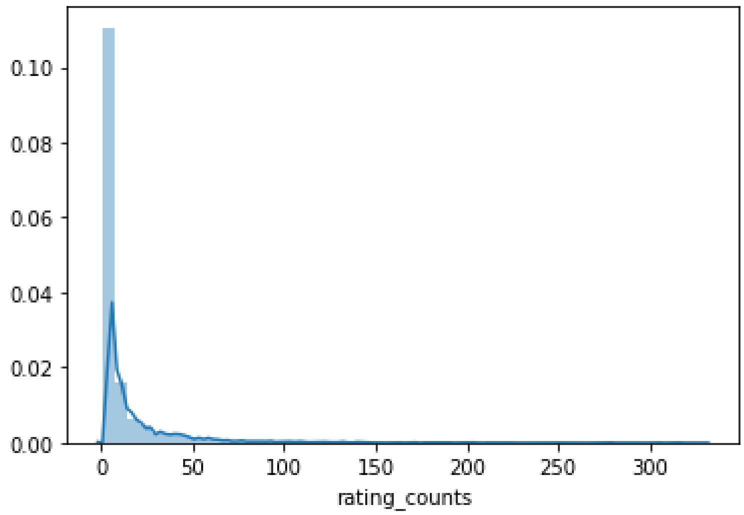
Out[41]:

title	rating	movieId	rating_counts
'71 (2014)	4.0	117867	1
'Hellboy': The Seeds of Creation (2004)	4.0	97757	1
'Round Midnight (1986)	3.5	26564	2
'Salem's Lot (2004)	5.0	27751	1
'Til There Was You (1997)	4.0	779	2

> Now I can see movie title, along with the average rating and number of ratings for the movie.

In [42]:
```
# plot distribution of users rating
sns.distplot(ratings_mean_count['rating_counts'])
```

Out[42]: <AxesSubplot:xlabel='rating_counts'>



From the output, I can conclude that most of the movies have received less than 50 ratings. While the number of movies having more than 100 ratings is very low.

In [43]:
```
q = ratings_mean_count['rating_counts'].quantile(0.95)  # 47

ratings_mean_count_filtered = ratings_mean_count[ratings_mean_count['ra
```

In [44]:
```
ratings_mean_count_filtered.sort_values('rating', ascending = False).he

```

Out[44]:

title	rating	movieId	rating_counts
Shawshank Redemption, The (1994)	4.429022	318	317
Godfather, The (1972)	4.289062	858	192
Fight Club (1999)	4.272936	2959	218
Cool Hand Luke (1967)	4.271930	1276	57
Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964)	4.268041	750	97
Rear Window (1954)	4.261905	904	84
Godfather: Part II, The (1974)	4.259690	1221	129
Departed, The (2006)	4.252336	48516	107
Goodfellas (1990)	4.250000	1213	126
Casablanca (1942)	4.240000	912	100

An interesting thing to note is that all the movies in the top 10 are older, this just could be because these movies have been around longer and have been rated more as a result

In [50]:
```
# define best films(movie ids) using
best_films_ids = ratings_mean_count_filtered.sort_values('rating', asce

# define a function which returns 5 best films(movie ids) based on aver
def cold_start(user, ratings, movies):

    return best_films_ids

```

Building the reccomender system

In [55]:

```
def recommend(userId, ratings, movies): # -> List[int]:
    user_favorites = ratings[ratings['userId'] == userId].sort_values('
    #  user_favorites contains id's of user's favourite movies
    answer = []

    rec_number = 5

    for movie_id in user_favorites:
        try:
            movie_descr = movies[all_genres].iloc[movie_id]  #  descrip
        except:
            print("problem:", movie_id)
            return cold_start(userId, ratings, movies)
        top_sim = dict() # {sim: index}

        lowest_top_sim = 0
        lowest_top_idx = 0

        for id, curr_description in movies[all_genres].iterrows():
            similarity, _ = pearsonr(movie_descr, curr_description)

            if len(top_sim) < rec_number:
                top_sim[similarity] = id
                continue

            #  update current top and check if we haven't added this fi
            min_top = min(list(top_sim.keys()))
            if similarity > min_top and id != movie_id and id not in an
                del top_sim[min_top]  # remove min of tops
                top_sim[similarity] = id  # add new
        answer += list(top_sim.items())

    return answer
```

In [56]:

```
np.random.seed(179)

# define a function which returns number of movies in recommendations t
def accuracy(ratings, movies, method):
    score = 0
    random_users = np.unique(np.random.choice(ratings['userId'].values,
    for user_id in tqdm(random_users):
        recs = method(user_id, ratings, movies)
        user_films = np.unique(ratings[ratings['userId'] == user_id]['m
        for r in recs:
            if r in user_films:
                score += 1
    return score
```

In [57]:
```
%%time

accuracy(ratings, movies, recommend)
```

```
HBox(children=(IntProgress(value=0, max=9), HTML(value='')))

{1.0: 173873, 0.7934920476158723: 6405, 0.8401680504168059: 184987, 0.793
4920476158722: 158830, 0.607843137254902: 188833}
{1.0: 193585, 0.6882472016116854: 188675, 0.6882472016116853: 193571, 0.5
461186812727503: 166291, -0.052631578947368425: 193609}
{1.0: 193579, 0.6882472016116854: 154975, 0.6882472016116853: 184257, 0.5
461186812727503: 58879, -0.052631578947368425: 193585}
problem: 116797

/Users/alinakorsunenko/opt/anaconda3/envs/learn-env/lib/python3.6/site-pa
ckages/ipykernel_launcher.py:11: DeprecationWarning: elementwise comparis
on failed; this will raise an error in the future.
  # This is added back by InteractiveShellApp.init_path()

{0.7934920476158721: 184349, 0.6882472016116853: 193585, 1.0: 190207, 0.7
934920476158722: 170355, -0.07647191129018727: 193609}
{1.0: 173873, 0.7934920476158723: 6405, 0.8401680504168059: 184987, 0.793
4920476158722: 158830, 0.607843137254902: 188833}
{0.7934920476158722: 189713, 0.6882472016116853: 193609, 1.0: 193571, 0.6
666666666666667: 109383}
{0.7934920476158723: 157172, 0.7934920476158722: 158830, 1.0: 3439, 0.840
1680504168059: 70697, 0.607843137254902: 160573}
{1.0: 193579, 0.6882472016116854: 154975, 0.6882472016116853: 184257, 0.5
461186812727503: 58879, -0.052631578947368425: 193585}
{1.0: 193585, 0.6882472016116854: 188675, 0.6882472016116853: 193571, 0.5
461186812727503: 166291, -0.052631578947368425: 193609}
{0.6882472016116853: 193571, 0.5461186812727503: 184349, 1.0: 193609, 0.6
882472016116854: 185435}
{0.6882472016116853: 193609, 0.6666666666666667: 178827, 1.0: 152970, 0.7
934920476158722: 188833}
problem: 69122
problem: 79428
{0.6882472016116854: 193585, 1.0: 188675, 0.7934920476158722: 189713, 0.6
666666666666667: 184931, -0.07647191129018725: 193609}
problem: 45503
{0.7934920476158723: 157172, 0.7934920476158722: 158830, 1.0: 3439, 0.840
1680504168059: 70697, 0.607843137254902: 160573}
{0.7934920476158722: 189713, 0.6882472016116853: 193609, 1.0: 193571, 0.6
666666666666667: 109383}
problem: 116897
{1.0: 193585, 0.6882472016116854: 188675, 0.6882472016116853: 193571, 0.5
461186812727503: 166291, -0.052631578947368425: 193609}
{0.8401680504168059: 153408, 0.8660254037844387: 45672, 1.0: 141818, 0.86
60254037844386: 120138, 0.8401680504168058: 158882}
{0.7934920476158721: 184349, 0.6882472016116853: 193585, 1.0: 190207, 0.7
934920476158722: 170355, -0.07647191129018727: 193609}

CPU times: user 1min 11s, sys: 1.36 s, total: 1min 13s
Wall time: 1min 20s
```

Out[57]: 12

In [54]:

```
%%time

accuracy(ratings, movies, cold_start)
```

```
HBox(children=(IntProgress(value=0, max=10), HTML(value='')))


CPU times: user 84.4 ms, sys: 8.46 ms, total: 92.9 ms
Wall time: 109 ms
```

Out[54]:

```
27
```

In [107]:

```
%%time

accuracy(ratings, movies, cold_start)
```

```
HBox(children=(IntProgress(value=0, max=77), HTML(value='')))
```

Out[107]:

```
191
```

In [108]:
```
%%time

accuracy(ratings, movies, recommend)
```

```
HBox(children=(IntProgress(value=0, max=78), HTML(value='')))

problem: 33794
problem: 27773
problem: 53123
problem: 60487
problem: 81845
problem: 27193
problem: 49272
problem: 128087
problem: 52281
problem: 79134
problem: 91529
problem: 30707
problem: 112852
problem: 135143
problem: 54997
problem: 44665
problem: 32587
problem: 104879
problem: 56782
problem: 30749
problem: 51931
problem: 138966
problem: 67255
problem: 79702
problem: 48394
problem: 37384
problem: 142422
problem: 46578
problem: 27773
problem: 26810
problem: 136469
problem: 48774
problem: 44761
problem: 54503
problem: 133419
problem: 33794
problem: 112552
```

Out[108]:
```
130
```

In [109]:
```
recommend(179, ratings, movies)
```

Out[109]:
```
[4, 360, 5, 77, 132]
```

In [110]:
```
ratings[ratings['userId'] == 179].sort_values('rating')['movieId']
```

Out[110]:
```
25915    339
25916    344
25914    329
25924    410
25925    420
        ...
25912    317
25913    318
25922    377
25894    110
25900    161
Name: movieId, Length: 69, dtype: int64
```

2. Singular Value Decomposition using Surprise

I will use the Surprise library that provides various ready-to-use powerful prediction algorithms (including (SVD) to evaluate its RMSE (Root Mean Squared Error) on the MovieLens dataset). It is a Python scikit building and analyzing recommender systems.

In [118]:
```
# Load Reader library
reader = Reader(line_format = 'user item rating timestamp', sep = '\t')

# Load ratings dataset with Dataset library
data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], r

# use this dataset to call cross_validate
baseline_results = cross_validate(BaselineOnly(), data, verbose = True,
```

```
Estimating biases using als...
Estimating biases using als...
Estimating biases using als...
Estimating biases using als...
Estimating biases using als...
Evaluating RMSE, MAE of algorithm BaselineOnly on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8786  0.8744  0.8688  0.8789  0.8623  0.8726  0.0063
MAE (testset)     0.6760  0.6722  0.6725  0.6771  0.6661  0.6728  0.0038
Fit time          0.22    0.22    0.24    0.23    0.27    0.24    0.02
Test time         0.12    0.12    0.19    0.12    0.13    0.13    0.03
```

In [125]:
```
knn_results = cross_validate(KNNBasic(), data, verbose = True, )
np.mean(knn_results['test_rmse'])
```

```
Computing the msd similarity matrix...
Done computing similarity matrix.
Computing the msd similarity matrix...
Done computing similarity matrix.
Computing the msd similarity matrix...
Done computing similarity matrix.
Computing the msd similarity matrix...
Done computing similarity matrix.
Computing the msd similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9445  0.9382  0.9485  0.9491  0.9425  0.9446  0.0040
MAE (testset)     0.7239  0.7171  0.7249  0.7292  0.7254  0.7241  0.0039
Fit time          0.13    0.14    0.21    0.16    0.15    0.16    0.03
Test time         1.75    1.66    1.81    1.68    1.70    1.72    0.05
```

Out[125]:
```
0.9445682981777115
```

In [127]:
```
# Use the SVD algorithm.
svd = SVD()

# Compute the RMSE of the SVD algorithm.
cross_validate(svd, data, measures=['RMSE', 'MAE', 'FCP'], cv=5, verbos
```

```
Evaluating RMSE, MAE, FCP of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8716  0.8754  0.8685  0.8698  0.8875  0.8746  0.0068
MAE (testset)     0.6742  0.6706  0.6654  0.6701  0.6799  0.6720  0.0048
FCP (testset)     0.6573  0.6637  0.6573  0.6580  0.6592  0.6591  0.0024
Fit time          6.41    7.90    6.57    5.86    7.69    6.89    0.78
Test time         0.27    0.18    0.24    0.24    0.18    0.22    0.04
```

Out[127]:
```
{'test_rmse': array([0.87155401, 0.87539026, 0.86853268, 0.8698431 , 0.88
745166]),
 'test_mae': array([0.67415955, 0.67058011, 0.66535117, 0.67007798, 0.679
92825]),
 'test_fcp': array([0.65725243, 0.66365786, 0.6572948 , 0.65802478, 0.659
16181]),
 'fit_time': (6.412759065628052,
  7.89998197555542,
  6.57332706451416,
  5.857029914855957,
  7.68517804145813),
 'test_time': (0.2718799114227295,
  0.18221807479858398,
  0.24133682250976562,
  0.23871183395385742,
  0.17577815055847168)}
```

I got a mean Root Mean Square Error of 0.87 which is pretty good. Let's now train on the dataset and arrive at predictions.

In [ ]:

```
1
```