Netflix Movie Reccomendation System

Business Understanding

Netflix is looking to improve their reccomendation system for new users. As part of a new trial membership program Netflix is looking to maximize their customer retention by providing the best possible reccomendations.

Netflix has attracted new users by using a free weekly trial membership. In order to maximize the number of customers that continue their membership, the reccomendations must be match the customers preferences. If the reccomendations are on point the customer is more likely to feel like there are enough options to continue the service past the free trial.

In [1]:
```
#initial imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Data

Import the four datasets to inspect and eventually combine into one dataframe for modeling.

The data is in the data folder:

- data/links.csv
- data/movies.csv
- data/ratings.csv
- data/tags.csv

Links dataframe

this dataframe will come in handy if we end up using additional data from imdb and the tmd for features in our model.

In [2]:
```
links = pd.read_csv('data/links.csv')
links.head()
```

Out[2]:

	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

In [3]:
```
links.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   movieId  9742 non-null   int64
 1   imdbId   9742 non-null   int64
 2   tmdbId   9734 non-null   float64
dtypes: float64(1), int64(2)
memory usage: 228.5 KB
```

Movies DataFrame

this contains the title and genre of the movies. The movieId column matches with our links dataframe. For example movieId 1 matches with movieId Toystory.

In [4]:
```
movies = pd.read_csv('data/movies.csv')
movies.head()
```

Out[4]:

	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 [5]:
```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   movieId  9742 non-null   int64 
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
dtypes: int64(1), object(2)
memory usage: 228.5+ KB
```

In [6]:
```
#extract the year of film from the title using regex to extract the year
movies['year'] = movies.title.str.extract(r'(?:\((\d{4})\))?\s*$', expand=F
movies.head()
```

Out[6]:

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

In [7]:
```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   movieId  9742 non-null   int64 
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
 3   year     9729 non-null   object
dtypes: int64(1), object(3)
memory usage: 304.6+ KB
```

In [8]:
```
#find the movies without years -
movies[movies.year.isna()]
```

Out[8]:

	movieId	title	genres	year
6059	40697	Babylon 5	Sci-Fi	NaN
9031	140956	Ready Player One	Action|Sci-Fi|Thriller	NaN
9091	143410	Hyena Road	(no genres listed)	NaN
9138	147250	The Adventures of Sherlock Holmes and Doctor W...	(no genres listed)	NaN
9179	149334	Nocturnal Animals	Drama|Thriller	NaN
9259	156605	Paterson	(no genres listed)	NaN
9367	162414	Moonlight	Drama	NaN
9448	167570	The OA	(no genres listed)	NaN
9514	171495	Cosmos	(no genres listed)	NaN
9515	171631	Maria Bamford: Old Baby	(no genres listed)	NaN
9518	171749	Death Note: Desu nôto (2006–2007)	(no genres listed)	NaN
9525	171891	Generation Iron 2	(no genres listed)	NaN
9611	176601	Black Mirror	(no genres listed)	NaN

In [9]:
```
# create dictionary to add the years for these movies
year_fix_dict ={40697:1998,
               140956:2018,
               143410:2015,
               147250:1939,
               149334:2016,
               156605:2016,
               162414:2002,
               167570:2016,
               171495:1996,
               171631:2017,
               171749:2006,
               171891:2013,
               176601:2013}
```

In [10]:
```
movies['year'] = movies['movieId'].map(year_fix_dict).fillna(movies['year']
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   movieId  9742 non-null   int64 
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
 3   year     9742 non-null   object
dtypes: int64(1), object(3)
memory usage: 304.6+ KB
```

In [11]:
```
movies.head()
```

Out[11]:

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

Ratings DataFrame

This dataframe contains userId, movieId, rating and a timestamp.

In [12]:
```
ratings = pd.read_csv('data/ratings.csv')
ratings.head()
```

Out[12]:

	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

In [13]:
```
ratings.rating.value_counts()
```

Out[13]:
```
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 [14]:
```
ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100836 entries, 0 to 100835
Data columns (total 4 columns):
 #   Column     Non-Null Count   Dtype  
---  ------     --------------   -----  
 0   userId     100836 non-null  int64  
 1   movieId    100836 non-null  int64  
 2   rating     100836 non-null  float64
 3   timestamp  100836 non-null  int64  
dtypes: float64(1), int64(3)
memory usage: 3.1 MB
```

In [15]:
```
ratings.movieId.value_counts()
```

Out[15]:
```
356       329
318       317
296       307
593       279
2571      278
         ... 
5986        1
100304      1
34800       1
83976       1
8196        1
Name: movieId, Length: 9724, dtype: int64
```

```
We have about 100000 ratings for roughly 10000 movies.
ratings range from .5 to 5.  4 is the most common rating.
```

Tags DataFrame

The tags dataframe has userId, movieId, tag and timestamp

In [16]:
```
tags = pd.read_csv('data/tags.csv')
tags.head()
```

Out[16]:

	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 [17]:
```
tags.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3683 entries, 0 to 3682
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   userId     3683 non-null   int64 
 1   movieId    3683 non-null   int64 
 2   tag        3683 non-null   object
 3   timestamp  3683 non-null   int64 
dtypes: int64(3), object(1)
memory usage: 115.2+ KB
```

Tags may be an important feature we will want to explore the tags and see if we can pinpoint some of the most used tags to add to our data

In [18]:
```
#tag value counts
tags.tag.value_counts()
```

Out[18]:
```
In Netflix queue        131
atmospheric              36
thought-provoking        24
superhero                24
Disney                   23
                       ... 
muppets                   1
android(s)/cyborg(s)      1
imaginary friend          1
investor corruption       1
Veterinarian              1
Name: tag, Length: 1589, dtype: int64
```

In [19]:
```
#create tag dictionary
keys = tags['tag'].value_counts(dropna=False).keys().tolist()
vals = tags['tag'].value_counts(dropna=False).tolist()
tag_dict = dict(zip(keys, vals))
tag_dict
```

Out[19]:
```
{'In Netflix queue': 131,
 'atmospheric': 36,
 'thought-provoking': 24,
 'superhero': 24,
 'Disney': 23,
 'funny': 23,
 'surreal': 23,
 'religion': 22,
 'sci-fi': 21,
 'dark comedy': 21,
 'quirky': 21,
 'psychology': 21,
 'suspense': 20,
 'twist ending': 19,
 'visually appealing': 19,
 'crime': 19,
 'politics': 18,
 'time travel': 16,
 'music': 16,
```

We may come back to the tags later.

Combined DataFrame

Below we will add the movie titles and genres to the ratings data to make a combined data frame

1. start the ratings dataframe and drop the timestamp.
2. use the movieId column to add the title and genre of the movie

In [20]:
```
#combined dataframe
#drop the timestamp column
df = ratings.drop('timestamp', axis=1)
#add title,genre and year using merge how=left will prevent more rows being
df = df.merge(movies, on='movieId', how='left')
df.head()
```

Out[20]:

	userId	movieId	rating	title	genres	year
0	1	1	4.0	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	1995
1	1	3	4.0	Grumpier Old Men (1995)	Comedy|Romance	1995
2	1	6	4.0	Heat (1995)	Action|Crime|Thriller	1995
3	1	47	5.0	Seven (a.k.a. Se7en) (1995)	Mystery|Thriller	1995
4	1	50	5.0	Usual Suspects, The (1995)	Crime|Mystery|Thriller	1995

In [21]:
```
df.shape
```

Out[21]:
```
(100836, 6)
```

In [22]:
```
df.describe()
```

Out[22]:

	userId	movieId	rating
count	100836.000000	100836.000000	100836.000000
mean	326.127564	19435.295718	3.501557
std	182.618491	35530.987199	1.042529
min	1.000000	1.000000	0.500000
25%	177.000000	1199.000000	3.000000
50%	325.000000	2991.000000	3.500000
75%	477.000000	8122.000000	4.000000
max	610.000000	193609.000000	5.000000

User ids range from 1-610. We will need to create new user ids that are outside of this range.

Further Data Exploration

In [23]:
```
#number of unique users
n_users = df.userId.nunique()
print(n_users,'users that have rated movies.')
##movies rated
mov_rat=df.movieId.nunique()
print(mov_rat,'different movies rated')
```

```
610 users that have rated movies.
9724 different movies rated
```

In [24]:
```
##top 20 best rated movies
agg_function = {'rating':['mean','count']}
movie_ratings = df.groupby(['movieId','title','genres']).agg(agg_function)
movie_ratings.sort_values(by=('rating','mean'), ascending=False)
```

Out[24]:

			rating	
			mean	count
movieId	title	genres		
88448	Paper Birds (Pájaros de papel) (2010)	Comedy|Drama	5.0	1
100556	Act of Killing, The (2012)	Documentary	5.0	1
143031	Jump In! (2007)	Comedy|Drama|Romance	5.0	1
143511	Human (2015)	Documentary	5.0	1
143559	L.A. Slasher (2015)	Comedy|Crime|Fantasy	5.0	1
...	...	...	...	...
157172	Wizards of the Lost Kingdom II (1989)	Action|Fantasy	0.5	1
85334	Hard Ticket to Hawaii (1987)	Action|Comedy	0.5	1
53453	Starcrash (a.k.a. Star Crash) (1978)	Action|Adventure|Fantasy|Sci-Fi	0.5	1
8494	Cincinnati Kid, The (1965)	Drama	0.5	1
71810	Legionnaire (1998)	Action|Adventure|Drama|War	0.5	1

9724 rows × 2 columns

We can see that we have many 5 rated movies as well as many .5 rated movies. It is good to know how many times each movie was rated as I have never heard of any of the movies that are currently listed at the top of the rating list. We have added the count to the agg function so now we can sort the movies by count.

In [25]:
```
movie_ratings.sort_values(by=('rating','count'), ascending=False)
```

Out[25]:

			rating	
			mean	count
movieId	title	genres		
356	Forrest Gump (1994)	Comedy|Drama|Romance|War	4.164134	329
318	Shawshank Redemption, The (1994)	Crime|Drama	4.429022	317
296	Pulp Fiction (1994)	Comedy|Crime|Drama|Thriller	4.197068	307
593	Silence of the Lambs, The (1991)	Crime|Horror|Thriller	4.161290	279
2571	Matrix, The (1999)	Action|Sci-Fi|Thriller	4.192446	278
...	...	...	...	...
4093	Cop (1988)	Thriller	1.500000	1
4089	Born in East L.A. (1987)	Comedy	2.000000	1
58351	City of Men (Cidade dos Homens) (2007)	Drama	4.000000	1
4083	Best Seller (1987)	Thriller	4.000000	1
193609	Andrew Dice Clay: Dice Rules (1991)	Comedy	4.000000	1

9724 rows × 2 columns

Now the movies at the top of the list are recognizable. Now we have an idea of movies that have been rated alot and most likely watched the most. This will be helpful when selecting movies for new users to rate. We want to only suggest movies that we currently have a good number of ratings for. This will make it more likely that they have seen the movie and it will make our model more useful because their will me users that have rated those movies.

Currently we have movies with anywhere from 1-329 rankings.

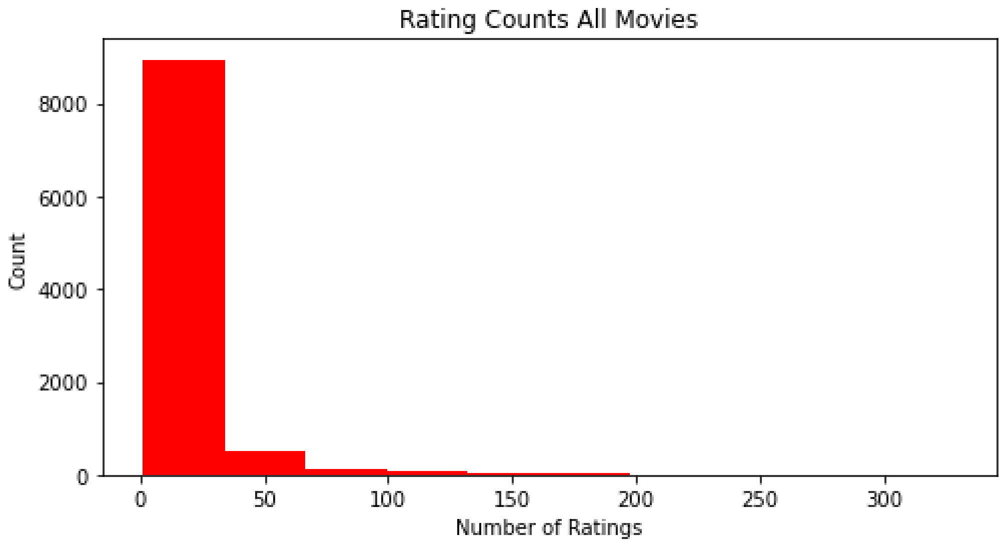
Finding the Best Movies for User Survey

It may be interesting to find movies that have a good balance amoung ratings. These movies may be better at pinpointing what a new user may like. for example movies that get mostly ratings of 4 or 5 may not tell us as much about a viewer as movies that recieve an equal amount of ratings from 1-5 or polarizing ratings. How do we do this...

For the sake of time we will limit the movies included in the user survey to movies that have atleast n ratings.

We can plot rating counts to see what a good number will be.

In [26]:
```
#histogram of rating count
fig, ax = plt.subplots(figsize=(8,4))
ax.hist(movie_ratings[('rating','count')],bins=10, color='red')
ax.set_title('Rating Counts All Movies')
ax.set_ylabel('Count')
ax.set_xlabel('Number of Ratings')
plt.show()
```



It looks like if we limit the movies for the survey to any movie that has more than 20 ratings we will have a good number of movies that the user has possibly seen and that enough other people have rated.

In [27]:
```
survey_movies = movie_ratings[movie_ratings[('rating','count')]>20]
survey_movies
```

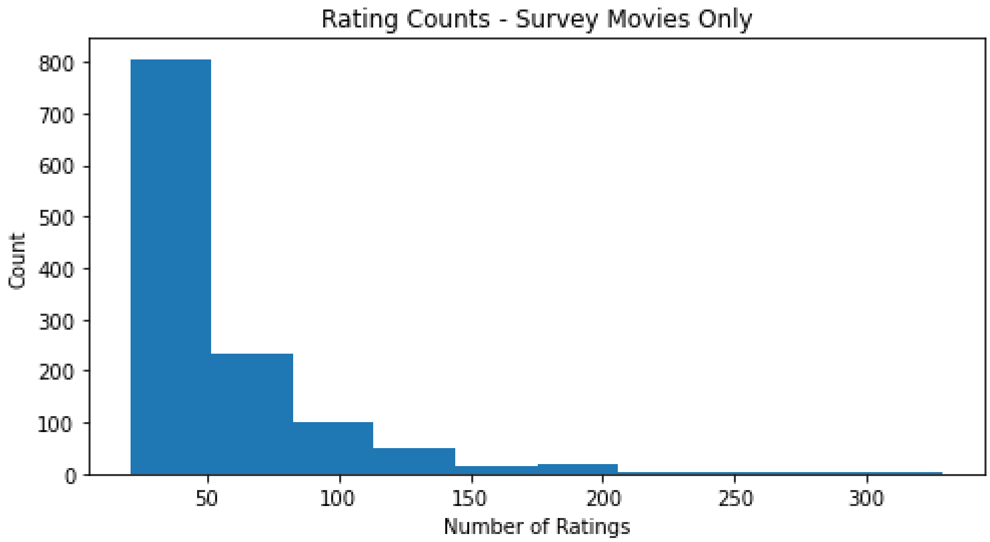
Out[27]:

			rating	
			mean	count
movieId	title	genres		
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	3.920930	215
2	Jumanji (1995)	Adventure|Children|Fantasy	3.431818	110
3	Grumpier Old Men (1995)	Comedy|Romance	3.259615	52
5	Father of the Bride Part II (1995)	Comedy	3.071429	49
6	Heat (1995)	Action|Crime|Thriller	3.946078	102
...	...	...	...	...
148626	Big Short, The (2015)	Drama	3.961538	26
152081	Zootopia (2016)	Action|Adventure|Animation|Children|Comedy	3.890625	32
164179	Arrival (2016)	Sci-Fi	3.980769	26
166528	Rogue One: A Star Wars Story (2016)	Action|Adventure|Fantasy|Sci-Fi	3.925926	27
168252	Logan (2017)	Action|Sci-Fi	4.280000	25

1235 rows × 2 columns

1235 movies will be included in our user rating survey.

In [28]:
```
#histogram of rating count
fig, ax = plt.subplots(figsize=(8,4))
ax.hist(survey_movies[('rating','count')],bins=10)
ax.set_title('Rating Counts - Survey Movies Only')
ax.set_ylabel('Count')
ax.set_xlabel('Number of Ratings')
plt.show()
```



Ratings Distribution Breakdown

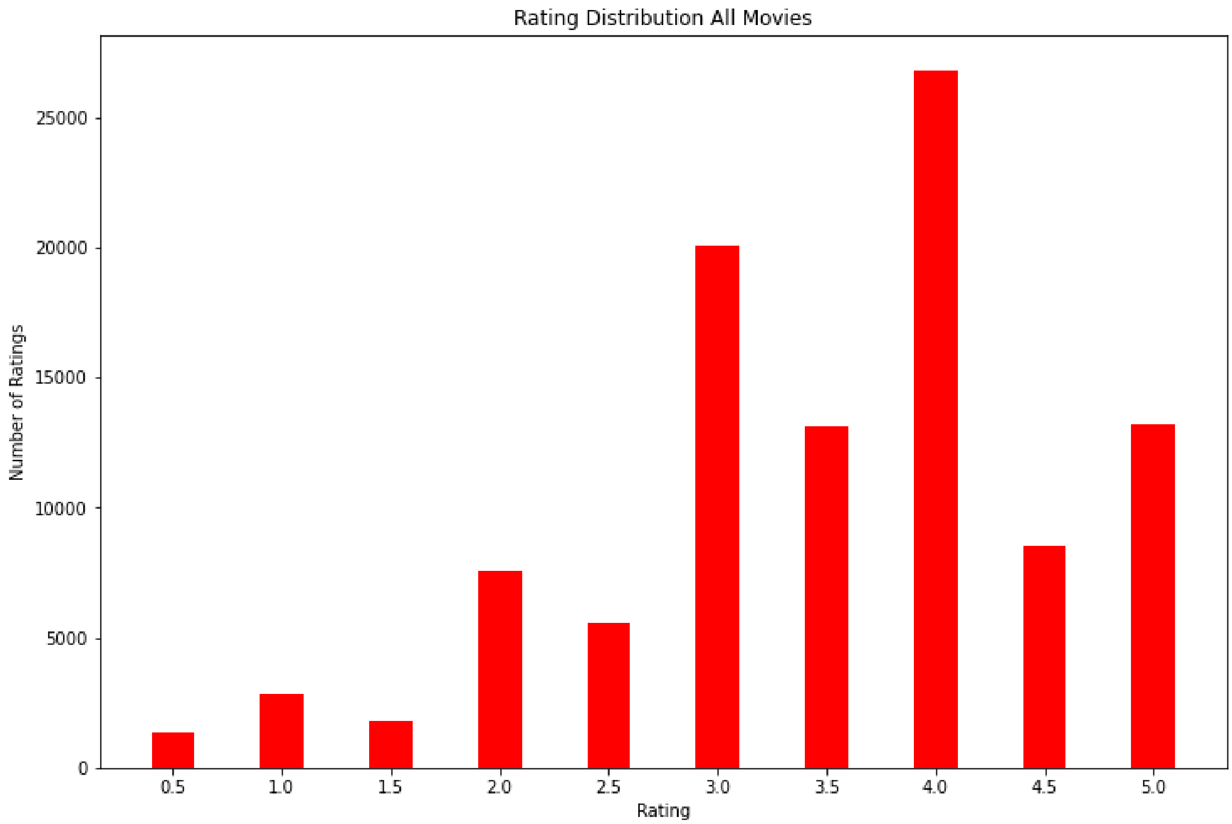
Visualizing the ratings.

In [29]:
```
rating_table = pd.DataFrame(df.groupby(['rating']).size(),columns=['Count']
rating_table
```

Out[29]:

	rating	Count
0	0.5	1370
1	1.0	2811
2	1.5	1791
3	2.0	7551
4	2.5	5550
5	3.0	20047
6	3.5	13136
7	4.0	26818
8	4.5	8551
9	5.0	13211

In [30]:
```
## ratings histogram
xs = rating_table['rating']
ys = rating_table['Count']
fig, ax = plt.subplots(figsize=(12,8))
ax.bar(xs,ys,tick_label=xs, width=0.2, color='red')
ax.set_title('Rating Distribution All Movies')
ax.set_ylabel('Number of Ratings')
ax.set_xlabel('Rating')
plt.show()
```



Movies By Year

In [31]:
```
movies.year.value_counts()
```

Out[31]:
```
2002      311
2006      295
2001      294
2007      284
2000      283
         ...
1998.0      1
1922        1
1915        1
1996.0      1
1939.0      1
Name: year, Length: 116, dtype: int64
```

In [32]:
```
movies['year'] = movies['year'].astype(int)
```

In [33]:
```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   movieId  9742 non-null   int64 
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
 3   year     9742 non-null   int64 
dtypes: int64(2), object(2)
memory usage: 304.6+ KB
```

In [34]:
```
max_year = movies.year.max()
min_year = movies.year.min()
print('Data includes movies from {} to {}'.format(min_year,max_year))
```

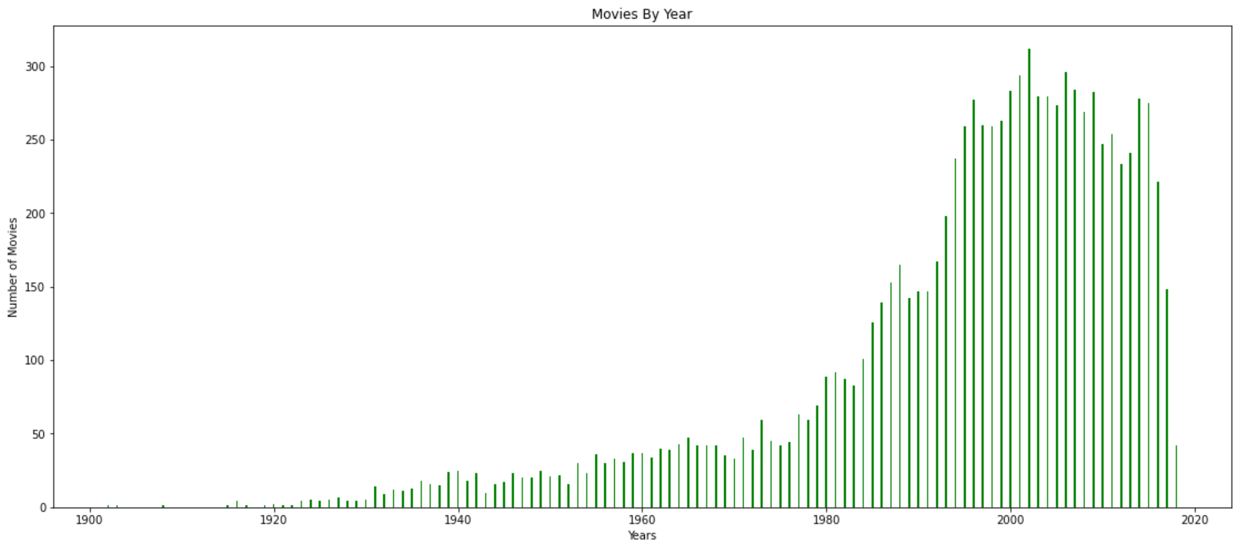
```
Data includes movies from 1902 to 2018
```

In [35]:
```
by_year = pd.DataFrame(movies.groupby(['year']).size(),columns=['Count']).r
by_year.head()
```

Out[35]:

	year	Count
0	1902	1
1	1903	1
2	1908	1
3	1915	1
4	1916	4

In [36]:
```
## ratings bar plot
xs = by_year['year']
ys = by_year['Count']
fig, ax = plt.subplots(figsize=(19,8))
ax.bar(xs,ys, width=0.2, color='green')
ax.set_title('Movies By Year')
ax.set_ylabel('Number of Movies')
ax.set_xlabel('Years')
plt.show()
```



I wonder if the year a movie was made or was rated has an influence on the average ratings.

Create User - Rating Matrix

We will create a matrix that has users and columns for each movie with that user ratings. This will be a very large sparse matrix. -- lots of zeros...

use df and pivot userId,movieId,rating

In [37]:
```
##create matrix from
model_matrix = df.pivot(index='userId',columns='movieId',values='rating').f
model_matrix.head()
```

Out[37]:

movieId	1	2	3	4	5	6	7	8	9	10	...	193565	193567	193571	193573	19357
userId																
1	4.0	0.0	4.0	0.0	0.0	4.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.
5	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.

5 rows × 9724 columns

In [38]:
```
model_matrix.shape
```

Out[38]:
```
(610, 9724)
```

In [39]:
```
non_zero = np.count_nonzero(model_matrix)
sparse_percentage = 1-(non_zero/(model_matrix.shape[0]*model_matrix.shape[1
print('matrix sparse percentage: {}%'.format(round(sparse_percentage *100))
```

```
matrix sparse percentage: 98%
```

As expected this is a sparse matrix will help in deciding which direction we will move in our iterative modeling process. Using the surprise library we will not need the model_matrix but it is interesting to see that our ratings data is 98% empty.

Surprise

We will import the needed tools from the Surprise libray below and begin our iterative modeling process.

In [40]:
```
from surprise import Reader, Dataset
from surprise.model_selection import cross_validate
from surprise.prediction_algorithms import SVD, KNNWithMeans, KNNBasic, KNN
from surprise.model_selection import GridSearchCV
```

We need our ratings data here. Lets make sure that we have the right data. We mostlikely need to drop the timestamp column still.

In [41]:
```
ratings.head()
```

Out[41]:

	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

In [42]:
```
# drop timestamp
rate_df = ratings.drop('timestamp', axis=1)
```

In [43]:
```
#create the surprise dataset
reader=Reader()
data=Dataset.load_from_df(rate_df,reader)
dataset=data.build_full_trainset()
dataset
```

Out[43]:
```
<surprise.trainset.Trainset at 0x7f7e3b73da30>
```

Lets explore the new surprise dataset to see if everything looks correct. We can look at the items and users to see how it compares to our original data.

In [44]:
```
#print out users and items
items = dataset.n_items
users = dataset.n_users
print('Users: {}\tItems: {}'.format(users,items))
```

```
Users: 610      Items: 9724
```

This matches with our matrix above. We are ready to model.

Iterative Modeling Process

For our modeling process we will begin with our baseline model. Because we have seen this data before we will start with our best parameters from a SVD model and then grid search around those values to see if we can do better.

Evaluation Metric -RMSE

RMSE - root mean square error RMSE was chosen as the metric to evaluate the models. This can be looked at as a regression problem and RMSE stays in the same scale as our data.

For example if a model scores 0.95, we know that our errors are roughly 1 full rating point off.

We can also pay attention to mean adjusted error (MAE), but I do not anticipate changing metrics at this point.

SVD

Singular Value Decomposition is a widely used dimensionality reduction tool.
In our previous work we found by using gridsearch that {'n_factors': 50, 'reg_all': 0.05} were the best parameters. We will run that first for our baseline model.

In [45]:
```
#svd baseline
baseline_model = SVD(n_factors=50,reg_all=0.05,random_state=42)
baseline_model.fit(dataset)
```

Out[45]:
```
<surprise.prediction_algorithms.matrix_factorization.SVD at 0x7f7e3b68eeb
0>
```

In [46]:
```
#cross-validate baseline model
baseline_cv = cross_validate(baseline_model,data,n_jobs=-1)
```

In [47]:
```
#print out results
for i in baseline_cv.items():
    print(i,'/n')
```

```
('test_rmse', array([0.872735  , 0.87653744, 0.87325638, 0.85901741, 0.86
87068 ])) /n
('test_mae', array([0.66774337, 0.67608562, 0.67213591, 0.66164877, 0.666
79016])) /n
('fit_time', (3.0863616466522217, 3.0278139114379883, 3.1797330379486084,
2.9170968532562256, 2.693882942199707)) /n
('test_time', (0.09775114059448242, 0.09209799766540527, 0.07855296134948
73, 0.07980990409851074, 0.08426618576049805)) /n
```

In [48]:
```
model_avg = np.mean(baseline_cv['test_rmse'])
model_avg
```

Out[48]:
```
0.8700506064811151
```

Create a Dictionary To Store Model Results

We want to store our model name and rmse in a dictionary to easily compare. We will also create a function to add further scores to our dictionary.

In [49]:
```
#score_dict will be used to store
score_dict={}
def add_to_dict(dict,model,score):
    dict[model]=score
    return dict
add_to_dict(score_dict,'baseline_model',model_avg)
```

Out[49]:
```
{'baseline_model': 0.8700506064811151}
```

SVD GridSearch

Lets see if we can improve on on our model by using a parameter grid search.

We used n_factors = 50(number of factors) and reg_all=0.05(regularization term) for these we will include values closer to these to test, because we had a wider range in our previous work.

lets include:

- n_epochs - The number of iterations default- 100
- lr_all – Parameter learning rate default - 0.005

Best Params

{'rmse': 0.8521706144532271, 'mae': 0.6524552323348267} {'rmse': {'n_factors': 60, 'reg_all': 0.075, 'n_epochs': 50, 'lr_all': 0.01}, 'mae': {'n_factors': 60, 'reg_all': 0.075, 'n_epochs': 50, 'lr_all': 0.01}}

*don't run the next cell to save time...

In [50]:
```
#svd gridsearch  -  This will take some time.  Be patient.

#params = {'n_factors':[40,50,60],
         #'reg_all':[0.025,0.05, 0.075],
         #'n_epochs':[25,50,100],
         #'lr_all':[0.0025,0.005,0.01]}
#svd_gs = GridSearchCV(SVD,param_grid=params,n_jobs=-1)
#svd_gs.fit(data)
```

In [51]:
```
#print(svd_gs.best_score)
#print(svd_gs.best_params)
```

Function to Fit and Get RMSE Scores From Model

The model_process function will perform the following steps:

1. Fit the model
2. Train the model
3. Cross Validate the model
4. Store the mean RMSE for the model

In [52]:
```
#our best model.
best_svd = SVD(n_factors=60, reg_all=0.075, lr_all=0.01, random_state=42)
```

In [53]:
```
##function
def model_process(model,name,train=dataset,full_data=data,dict=score_dict):
    '''
    model- actual model
    name - string of model name for storing in dictionary
    train - training data
    full_data - all of the data
    dict - scoring dictionary

    '''
    #fit the model
    model.fit(train)

    #cross-validate the model
    model_cv = cross_validate(model,full_data,n_jobs=-1,cv=5,verbose=True)

    #score RMSE
    rmse = np.mean(model_cv['test_rmse'])

    #add to score dictionary
    add_to_dict(dict,name,rmse)

    return dict
```

In [54]:
```
#our best svd model just changes the reg_all. the others were default setti
best_svd = SVD(n_factors=60, reg_all=0.075,lr_all=0.01, random_state=42)
#fit, score and add to dictionary using function
model_process(best_svd,'best_svd',dataset,data,score_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8610  0.8559  0.8621  0.8623  0.8634  0.8610  0.0026
MAE (testset)     0.6614  0.6586  0.6610  0.6593  0.6637  0.6608  0.0018
Fit time          3.69    4.27    3.98    3.68    3.55    3.83    0.26
Test time         0.10    0.10    0.11    0.11    0.09    0.10    0.01
```

Out[54]:
```
{'baseline_model': 0.8700506064811151, 'best_svd': 0.8609507732753328}
```

This is only a slight improvement in our model. Rememeber that our rating scale is 1-5. So we are still off by about .86 of rating point.

Below we will try some other models

KNN Algorithms

We can check some other algorithms to see if we can do better and to make sure that we have the best model possible.

lets compare

- KNNBasic
- KNNBaseline

- KNNWithMeans
- KNNWithZScore

We can do gridsearch with these to see if we can do better.

KNNBasic

basic KNN model

In [55]:
```
##KNNBasic
knn_basic = KNNBasic(sim_options={'name': 'pearson', 'user_based': True},ra
model_process(knn_basic,'knn_basic')
```

```
Computing the pearson 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)    0.9808  0.9744  0.9695  0.9668  0.9693  0.9721  0.0050
MAE (testset)     0.7579  0.7527  0.7496  0.7451  0.7496  0.7510  0.0042
Fit time          0.32    0.45    0.48    0.42    0.32    0.40    0.07
Test time         1.25    1.09    1.06    1.04    1.04    1.09    0.08
```

Out[55]:
```
{'baseline_model': 0.8700506064811151,
 'best_svd': 0.8609507732753328,
 'knn_basic': 0.9721431917197332}
```

KNNBaseline

KNN that takes a baseline rating into account.

In [56]:
```
knn_baseline = KNNBaseline(sim_options={'name': 'pearson', 'user_based': Tr
model_process(knn_baseline,'knn_baseline')
```

```
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBaseline on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8753  0.8770  0.8793  0.8738  0.8812  0.8773  0.0027
MAE (testset)     0.6691  0.6689  0.6705  0.6663  0.6747  0.6699  0.0028
Fit time          0.40    0.45    0.45    0.38    0.36    0.41    0.04
Test time         1.51    1.40    1.48    1.42    1.43    1.45    0.04
```

Out[56]:
```
{'baseline_model': 0.8700506064811151,
 'best_svd': 0.8609507732753328,
 'knn_basic': 0.9721431917197332,
 'knn_baseline': 0.8773273471611617}
```

KNNWithMeans

Takes into account mean rating for each user

In [57]:
```
#KNNWithMeans
knn_wm = KNNWithMeans(sim_options={'name': 'pearson', 'user_based': True},r
model_process(knn_wm,'knn_wm')
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9016  0.8873  0.8958  0.8957  0.8957  0.8952  0.0045
MAE (testset)     0.6864  0.6769  0.6841  0.6827  0.6793  0.6819  0.0034
Fit time          0.32    0.29    0.32    0.31    0.31    0.31    0.01
Test time         1.12    1.15    1.11    1.11    1.09    1.12    0.02
```

Out[57]:
```
{'baseline_model': 0.8700506064811151,
 'best_svd': 0.8609507732753328,
 'knn_basic': 0.9721431917197332,
 'knn_baseline': 0.8773273471611617,
 'knn_wm': 0.8952198388185757}
```

KNNWithZScore

Takes into account the z-score normalization of each user.

In [58]:
```
knn_wzs = KNNWithZScore(sim_options={'name': 'pearson', 'user_based': True}
model_process(knn_wzs,'knn_wzs')
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8973  0.8987  0.8831  0.8900  0.8964  0.8931  0.0058
MAE (testset)     0.6769  0.6803  0.6667  0.6726  0.6779  0.6749  0.0048
Fit time          0.34    0.35    0.34    0.34    0.33    0.34    0.01
Test time         1.28    1.28    1.27    1.25    1.21    1.26    0.03
```

Out[58]:
```
{'baseline_model': 0.8700506064811151,
 'best_svd': 0.8609507732753328,
 'knn_basic': 0.9721431917197332,
 'knn_baseline': 0.8773273471611617,
 'knn_wm': 0.8952198388185757,
 'knn_wzs': 0.8931068970179382}
```

the knn_baseline model was the best of the 3 and just a little bit higher than our best svd model. We can try to tune that model to see if we can improve the performance.

KNNBaseline HyperTuning

In [59]:
```
#KNNBaseline with more parameters
knn_baseline_min_k_5 = KNNBaseline(min_k=5,sim_options={'name': 'pearson',
model_process(knn_baseline_min_k_5,'knn_baseline_min_k_5')
```

```
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBaseline on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8666  0.8637  0.8710  0.8624  0.8684  0.8664  0.0031
MAE (testset)     0.6630  0.6622  0.6676  0.6602  0.6639  0.6634  0.0024
Fit time          0.36    0.36    0.34    0.33    0.34    0.35    0.01
Test time         1.43    1.46    1.44    1.43    1.40    1.43    0.02
```

Out[59]:
```
{'baseline_model': 0.8700506064811151,
 'best_svd': 0.8609507732753328,
 'knn_basic': 0.9721431917197332,
 'knn_baseline': 0.8773273471611617,
 'knn_wm': 0.8952198388185757,
 'knn_wzs': 0.8931068970179382,
 'knn_baseline_min_k_5': 0.8664105966178457}
```

In [60]:
```
#KNNBaseline with more parameters
knn_baseline_k_30 = KNNBaseline(min_k=30,sim_options={'name': 'pearson', 'u
model_process(knn_baseline_k_30,'knn_baseline_k_30')
```

```
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBaseline on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8585  0.8678  0.8744  0.8702  0.8639  0.8670  0.0054
MAE (testset)     0.6617  0.6674  0.6706  0.6649  0.6652  0.6660  0.0029
Fit time          0.35    0.36    0.37    0.39    0.39    0.37    0.02
Test time         1.48    1.42    1.46    1.44    1.42    1.44    0.03
```

Out[60]:
```
{'baseline_model': 0.8700506064811151,
 'best_svd': 0.8609507732753328,
 'knn_basic': 0.9721431917197332,
 'knn_baseline': 0.8773273471611617,
 'knn_wm': 0.8952198388185757,
 'knn_wzs': 0.8931068970179382,
 'knn_baseline_min_k_5': 0.8664105966178457,
 'knn_baseline_k_30': 0.866957860420207}
```

These both slightly improved our RMSE. It may warrant taking the time to run a gridsearch with different values of k, min_k

GridSearchCV with KNNBaseline

In [61]:
```
#this will take a minute or so....
#params = {'k': [15,30,40],
          #'min_k': [1,3,5]}
#kb = KNNBaseline(sim_options = {'name': 'pearson', 'user_based': True})
#knnbaseline_gs = GridSearchCV(KNNBaseline,param_grid=params)
#knnbaseline_gs.fit(data)
```

In [62]:
```
#Get the scores and best params
#print(knnbaseline_gs.best_params)
#print(knnbaseline_gs.best_score)
```

In [63]:
```
#best KNN Baseline model
knn_baseline_best = KNNBaseline(k=30,min_k=5,sim_options={'name': 'pearson'
model_process(knn_baseline_best,'knn_baseline_best')
```

```
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBaseline on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8661  0.8621  0.8682  0.8692  0.8646  0.8660  0.0025
MAE (testset)     0.6635  0.6603  0.6652  0.6653  0.6637  0.6636  0.0018
Fit time          0.47    0.49    0.54    0.69    0.60    0.56    0.08
Test time         1.91    1.90    1.83    1.53    1.46    1.73    0.19
```

Out[63]:
```
{'baseline_model': 0.8700506064811151,
 'best_svd': 0.8609507732753328,
 'knn_basic': 0.9721431917197332,
 'knn_baseline': 0.8773273471611617,
 'knn_wm': 0.8952198388185757,
 'knn_wzs': 0.8931068970179382,
 'knn_baseline_min_k_5': 0.8664105966178457,
 'knn_baseline_k_30': 0.866957860420207,
 'knn_baseline_best': 0.8660247232095186}
```

Final Model Selection

Our best_svd model has the best results. The RMSE score for this model was 0.8609507732753328. When trying other models their optimized settings were all trending back towards this value but never quite reaching it. This is most likely the best that we can do with the given data. As more data becomes available this could possibly change.

Making Predicitons

Below will test code for predictions before building a function. Need to determine the best way to get the actual movie data out of the surpries formatted predictions.

In [64]:
```
best_svd.predict(12,12)
```

Out[64]:
```
Prediction(uid=12, iid=12, r_ui=None, est=3.627661427518636, details={'wa
s_impossible': False})
```

In [65]:
```
#making predictions for user 10 movie 1
pred = best_svd.predict(10,1)
pred
```

Out[65]:
```
Prediction(uid=10, iid=1, r_ui=None, est=3.4843803919078966, details={'wa
s_impossible': False})
```

In [66]:
```
#use the movies dataframe to get actual movie information for reccomendatio
movies.head()
```

Out[66]:

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

In [67]:
```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   movieId  9742 non-null   int64
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
 3   year     9742 non-null   int64
dtypes: int64(2), object(2)
memory usage: 304.6+ KB
```

Displaying Movies

Need to use the movies dataframe to get the information for the movie. The surprise format gives us the the item id (iid) which can be matched to the index of our movies dataframe.

Use .loc to find the index that matches and return the title and genre

pred[0] - userId, pred[1] - index, pred[2] - actual user rating pred[3] - predicted rating

In [68]:
```
#retrieve movie information from our prediction
title = movies.loc[pred[1]]['title']
genre = movies.loc[pred[1]]['genres']
pred_rating=round(pred[3],1)
print('Title: {}\nGenre: {}\nProjected Rating: {}'.format(title,genre,pred_
```

```
Title: Jumanji (1995)
Genre: Adventure|Children|Fantasy
Projected Rating: 3.5
```

Find n Movie Reccomendations For Specific User

Below we will create a function that will return n movies for a specific user.

In [69]:
```
#get number of items
dataset.n_items
```

Out[69]:
```
9724
```

In [70]:
```
#import heapq for getting top items from list
import heapq
#import regex
import re

#define reccomendation function
def n_movies_rec(user,model, movie_list,n=5):

    '''
    this will return n number of movies for a specific user.

    user - which user
    model - model to make the predictions
    n- number of reccomendations(default 5)
    movie_list -df of movies to pick from -  default(movies(all of the item
    '''
    #change display text based on user choices
    rec_title = 'YOUR CUSTOMIZED RECOMENDATIONS'
    if movie_list is movies:
        rec_title = 'STANDARD RECOMENDATIONS'

    display_movies = []
    #get the items from the dataset
    total_items = dataset.n_items


    #list for movie ratings
    movie_ratings = []


    #create a list of movieId's
    ids_list = pd.unique(movie_list['movieId'])


    #populate movie_ratings for raw list
    for item in range(total_items):
        #append the movie to the movie_rating list if it is in the movie_li
        movie_ratings.append(model.predict(user,item))

    #remove any movies from movie_ratings not in movie list before getting
    final_ratings=[]

    for mov in movie_ratings:
        mov_id = movies.iloc[mov[1]]['movieId']
        if mov_id in ids_list:
            #add movie to the final_ratings
            final_ratings.append(mov)


    print('final ratings include {} ratings out of {}'.format(len(final_rat
    print('-------------------------------------------------')
    print('-------------------------------------------------')
    print(' ')
    print(rec_title)
    print(' ')
```

```
    #use heapq.nlagerst to get the N highest rated movies. the 3 item of ea
    raw_list = heapq.nlargest(n,final_ratings,key=lambda x:x[3])
    # get movie info from movies dataframe

    for p in raw_list:
        title = movies.iloc[p[1]]['title']
        genre = movies.iloc[p[1]]['genres']
        pred_rating=round(p[3],1)
        print('Title: {}\nGenre: {}\nProjected Rating: {}'.format(title,gen
        print('\n')
    return None

#test the function for 10 movies for user 110
n_movies_rec(110,best_svd, movies, 10)
```

```
final ratings include 9724 ratings out of 9724
-------------------------------------------------
-------------------------------------------------

STANDARD RECOMENDATIONS

Title: My Best Friend's Wedding (1997)
Genre: Comedy|Romance
Projected Rating: 4.4


Title: Marat/Sade (1966)
Genre: Drama|Musical
Projected Rating: 4.4


Title: Virtuosity (1995)
Genre: Action|Sci-Fi|Thriller
Projected Rating: 4.4


Title: RocketMan (a.k.a. Rocket Man) (1997)
Genre: Children|Comedy|Romance|Sci-Fi
Projected Rating: 4.4


Title: I Love Trouble (1994)
Genre: Action|Comedy
Projected Rating: 4.4


Title: Escape from New York (1981)
Genre: Action|Adventure|Sci-Fi|Thriller
Projected Rating: 4.4


Title: Cement Garden, The (1993)
Genre: Drama
Projected Rating: 4.3


Title: Withnail & I (1987)
Genre: Comedy
```

```
Projected Rating: 4.3


Title: Amistad (1997)
Genre: Drama|Mystery
Projected Rating: 4.3


Title: Hoodlum (1997)
Genre: Crime|Drama|Film-Noir
Projected Rating: 4.3
```

In [71]:
```
#try out another user to make sure this is working and giving different rec
n_movies_rec(8,best_svd,movies, 10)
```

```
final ratings include 9724 ratings out of 9724
-------------------------------------------------
-------------------------------------------------

STANDARD RECOMENDATIONS

Title: I Love Trouble (1994)
Genre: Action|Comedy
Projected Rating: 4.5


Title: Marat/Sade (1966)
Genre: Drama|Musical
Projected Rating: 4.5


Title: Cement Garden, The (1993)
Genre: Drama
Projected Rating: 4.5


Title: Escape from New York (1981)
Genre: Action|Adventure|Sci-Fi|Thriller
Projected Rating: 4.4


Title: My Best Friend's Wedding (1997)
Genre: Comedy|Romance
Projected Rating: 4.4


Title: Lady Vengeance (Sympathy for Lady Vengeance) (Chinjeolhan geumjass
i) (2005)
Genre: Crime|Drama|Mystery|Thriller
Projected Rating: 4.4


Title: Star Wars: Episode V - The Empire Strikes Back (1980)
Genre: Action|Adventure|Sci-Fi
Projected Rating: 4.4


Title: Uncle Buck (1989)
Genre: Comedy
Projected Rating: 4.4


Title: Indian Summer (a.k.a. Alive & Kicking) (1996)
Genre: Comedy|Drama
Projected Rating: 4.4


Title: Pompatus of Love, The (1996)
Genre: Comedy|Drama
```

```
Projected Rating: 4.4
```

This will give us reccomendations for existing users. We will then need to add by category and beable to create a new user and provide results

Obtain New User Ratings

We will create a function to obtain new user ratings. We will want to gather at least 5 ratings in order to make reccomendations from our model. The user will be given movies to rate on a scale of 0.5 to 5. They should have the option to skip a movie if they have not seen it.

- use survey_movies dataframe for this function. This should make it more likely that the new user has seen the movie

We need to be able to do the following:

- get new user ratings
- add ratings to the ratings 'data'
- read the data into a suprise dataset
- train the model
- return the predicitons (5)

We will create funcitons to work through this process and then create a master function to complete the entire process.

Get New User Ratings

We need to use the original rate_df to add the ratings of the new user.

In [72]:
```
rate_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100836 entries, 0 to 100835
Data columns (total 3 columns):
 #   Column   Non-Null Count   Dtype  
---  ------   --------------   -----  
 0   userId   100836 non-null  int64  
 1   movieId  100836 non-null  int64  
 2   rating   100836 non-null  float64
dtypes: float64(1), int64(2)
memory usage: 2.3 MB
```

In [73]:
```
np.max(rate_df.userId)
```

Out[73]:
```
610
```

In [74]:
```
sm_df = survey_movies.drop('rating',axis=1)
```

In [75]:
```
sm_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 1235 entries, (1, 'Toy Story (1995)', 'Adventure|Animation|Ch
ildren|Comedy|Fantasy') to (168252, 'Logan (2017)', 'Action|Sci-Fi')
Empty DataFrame
```

In [76]:
```
sm_df = sm_df.reset_index()
```

In [77]:
```
sm_df
```

Out[77]:

	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	5	Father of the Bride Part II (1995)	Comedy
4	6	Heat (1995)	Action|Crime|Thriller
...	...	...	...
1230	148626	Big Short, The (2015)	Drama
1231	152081	Zootopia (2016)	Action|Adventure|Animation|Children|Comedy
1232	164179	Arrival (2016)	Sci-Fi
1233	166528	Rogue One: A Star Wars Story (2016)	Action|Adventure|Fantasy|Sci-Fi
1234	168252	Logan (2017)	Action|Sci-Fi

1235 rows × 3 columns

New User Rating Function

The new user rating function selects movies to be rated and returns a new dataframe of that users ratings.

In [78]:
```
def new_user_ratings(movies,n,genre=None):
    '''
    movies - selected movie dataframe
    n - (int) number of movies
    genre- (str) can specify a genre to further limit the data

    '''

    #narrow down the movies if genre is selected
    to_be_rated = movies
    if genre:
        to_be_rated = movies[movies['genres'].str.contains(genre)]

    #create new user id one higher than the max userId value
    user_id = np.max(rate_df.userId)+1

    #list to store the new ratings
    ratings = []

    #use while loop to get n ratings from our survey_movies
    while n > 0:
        # select a sample movie
        movie = to_be_rated.sample(1)
        #clean up the presentation of the movie for the survey
        title = movie.title.to_string()
        title = re.sub("[^a-zA-Z0-9(),'']+", " ", title)
        print(title)
        rating = input("Rate the movie from 1-5.  enter 'x' if you haven't
        # make sure user enters an acceptable value
        if rating not in ['1','2','3','4','5']:
            continue
        else:
            #need to use column names from rate_df
            rated = {'userId':user_id,'movieId':movie['movieId'].values[0],
            ratings.append(rated)
            n -=1

    return pd.DataFrame(ratings)
```

Uncomment the code below to test out the new user function. It is commented out so movies do not need to be rated each time the notebook is restarted and cells are run.

In [94]:
```
#test out the ratings
#new_ratings = new_user_ratings(sm_df,5)
#new_ratings
```

...

Add Ratings to the Original Ratings Data

We want to add these to rate_df

In [80]:
```
rate_df.head()
```

Out[80]:

	userId	movieId	rating
0	1	1	4.0
1	1	3	4.0
2	1	6	4.0
3	1	47	5.0
4	1	50	5.0

Add Ratings Function

This functions will add the new user ratings to the original rating data.

It will return the data in both surprise and pandas form.

In [81]:
```
#define function to add movies to the Data - read into surprise
def add_ratings(new_rate, existing_data):
    existing_data = pd.concat([new_rate, existing_data],ignore_index=True)
    updated_ratings = Dataset.load_from_df(existing_data,reader)
    rate_df = existing_data
    #return both the dataframe and surprise dataframe
    return existing_data,updated_ratings
```

Fit Model

best_svd = SVD(n_factors=60, reg_all=0.075, lr_all=0.01, random_state=42)

use the new add_ratings function to fit the the model

In [82]:
```
#fit the model using the add_ratings function
best_svd = SVD(n_factors=60, reg_all=0.075, lr_all=0.01, random_state=42)
ratings,surprise_ratings = add_ratings(new_ratings,rate_df)
best_svd.fit(surprise_ratings.build_full_trainset())
```

Out[82]:
```
<surprise.prediction_algorithms.matrix_factorization.SVD at 0x7f7e1971110
0>
```

In [83]:
```
ratings.describe()
```

Out[83]:

	userId	movieId
count	100841.000000	100841.000000
mean	326.141688	19435.505985
std	182.624980	35531.388189
min	1.000000	1.000000
25%	177.000000	1199.000000
50%	325.000000	2991.000000
75%	477.000000	8121.000000
max	611.000000	193609.000000

Return Predictions

use 'n_movies_rec' function to return 5 movies.

In [84]:
```
n_movies_rec(np.max(ratings.userId),best_svd,movies,5)
```

```
final ratings include 9724 ratings out of 9724
-------------------------------------------------
-------------------------------------------------

STANDARD RECOMENDATIONS

Title: Marat/Sade (1966)
Genre: Drama|Musical
Projected Rating: 4.3


Title: Love Potion #9 (1992)
Genre: Comedy|Romance
Projected Rating: 4.2


Title: Cement Garden, The (1993)
Genre: Drama
Projected Rating: 4.2


Title: I Love Trouble (1994)
Genre: Action|Comedy
Projected Rating: 4.2


Title: Barefoot Contessa, The (1954)
Genre: Drama
Projected Rating: 4.2
```

Put it all Together

Function to get some user preferences. This is being added to narrow down the types of movies being reccomended. We can ask the user a few questions to limit the pool of movies using year and genre. the function should return a dataframe that can be used in the user_survey() function

In [85]:
```
movies[movies['year']>1980]
```

Out[85]:

	movieId	title	genres	year
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	1995
1	2	Jumanji (1995)	Adventure|Children|Fantasy	1995
2	3	Grumpier Old Men (1995)	Comedy|Romance	1995
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance	1995
4	5	Father of the Bride Part II (1995)	Comedy	1995
...	...	...	...	...
9737	193581	Black Butler: Book of the Atlantic (2017)	Action|Animation|Comedy|Fantasy	2017
9738	193583	No Game No Life: Zero (2017)	Animation|Comedy|Fantasy	2017
9739	193585	Flint (2017)	Drama	2017
9740	193587	Bungo Stray Dogs: Dead Apple (2018)	Action|Animation	2018
9741	193609	Andrew Dice Clay: Dice Rules (1991)	Comedy	1991

8092 rows × 4 columns

Genre List

need to make a list of genres that users can select or eliminate

In [86]:
```
##go through genres column and create a list of genres
#list to hold the genres
genre_list = []
#function to get all the genres separated and added to the list
def get_genre(row):
    words = row.split('|')
    for w in words:
        w = w.lower()
        if w not in genre_list:
            genre_list.append(w)

#lambda function to get the entire dataframe
movies['genres'].map(lambda x: get_genre(x))
genre_list.remove('(no genres listed)')
genre_list
```

Out[86]:
```
['adventure',
 'animation',
 'children',
 'comedy',
 'fantasy',
 'romance',
 'drama',
 'action',
 'crime',
 'thriller',
 'horror',
 'mystery',
 'sci-fi',
 'war',
 'musical',
 'documentary',
 'imax',
 'western',
 'film-noir']
```

Questionnaire

The questionnaire will allow the user to limit the range of years. -- may need to build in something to prevent bad inputs and make sure that the user cannot limit the data beyound the output of 5 movies.

Then the questionnaire will allow the user to include only certain genres.

You can uncomment the last line to just run the questionnaire if needed.

In [95]:
```
def questionnaire():
    """
    this funciton will take the movies dataframe and return an custom dataf
    questions.

    """
    # year  -  change the range of the movies
    print('Our movie library contains films from {} to {}.'.format(int(min_
    year_range=input("Type 'yes' if you would like to narrow the range of y

    if year_range == 'yes':
        oldest = int(input('Enter the first year of your desired range? '))
        newest = int(input('Enter the last year of your desired range? '))
        custom_movies = movies[(movies['year'] >= oldest ) & (movies['year'
    else:
        custom_movies = movies

    # genres -  limit the genres included in recs

    print(' ')
    print('Our movie library contains films from the following genres: ')
    print(' ')
    print(' ')

    print(genre_list)

    #ask if they want to modify
    limit_genre = input("Type 'yes' if you would like to limit the genres:

    mod_genre = []

    # check if they want to limit genres
    if limit_genre == 'yes':
        print('')
        print("Enter any genres that you would like to include in your recc
        print("leave blank and hit enter to stop")

        #continue looping until they don't enter anything
        while True:
            word = input()
            if word:
                mod_genre.append(word.lower())
            else:
                break



    #change custom movies to remove movies if they are not in the mod_genre

        custom_movies = custom_movies[custom_movies['genres'].map(
            lambda x: any(substring in x.lower() for substring in mod_genre

    return custom_movies

#questionnaire()
```

Below we will create a new function to run all of this on account creation.

Limit the rate_df by user selection

Testing out code for the function below. When the questionnaire was added it created a need to update other parts of the function. This works but I'd rather have all the rating data and limit the predictions afterwards

In [88]:
```
##rate_df needs to match the custom_movies dataframe created after the ques
id_ins = [2291,55269]
testing_df = rate_df[rate_df['movieId'].isin(id_ins)]
testing_df
```

Out[88]:

	userId	movieId	rating
146	1	2291	5.0
1923	18	2291	4.0
5163	33	2291	2.0
6949	47	2291	2.5
7573	51	2291	4.0
...	...	...	...
96627	603	2291	4.0
97215	605	2291	3.5
97743	606	2291	4.0
99032	608	2291	3.5
100277	610	55269	4.0

101 rows × 3 columns

This worked, but it will probably be easier to limit the movies in the n_rec_movies function.

User Survey Function

The user survey is the function that will take a user through the entire process.

User will rate a selected number of movies ratings will will be added to the the rate_df model will be created and fit the questionnaire will limit the pool of movies reccomendations will be produced from the highest projected ratings from the limited pool

In [89]:
```
def user_survey():
    '''
    completes the entire process of rating the movies and running the model
    displaying the predictions
    '''
    # make rate_df global var to be able to update inside of the function
    global rate_df
    #use updated dataframe if it is longer than rate_df-- it will be after

    # How many movies do you want to rate?
    movie_count = int(input('How many movies would you like to rate? (enter
    while movie_count < 1:
        movie_count = int(input('How many movies would you like to rate? (e

    # get ratings - gets list of the rated movies
    new_ratings = new_user_ratings(sm_df,movie_count)

    #create model
    best_svd = SVD(n_factors=60, reg_all=0.075, lr_all=0.01, random_state=4

    #update movie data and create new surprise data
    rate_data, surprise_ratings = add_ratings(new_ratings,rate_df)

    #fit the model
    dataset = surprise_ratings.build_full_trainset()
    best_svd.fit(dataset)

    #produce reccomendations
    #get the correct user
    user = np.max(rate_data.userId)

    #1. unfiltered reccomendations
    unfiltered = n_movies_rec(user,best_svd,movies,5)

    #2. Trim down the movie pool for reccomendations based on user question
    update_list = questionnaire()

    #the current user is the one with higest userId

    #gets the customized recs

    personalized = n_movies_rec(user,best_svd,update_list,5)

    #update the rate_df
    rate_df = rate_data

    return None
```

Testing the user survey to make sure all of the parts are working. Code is commented out to prevent the survey from having to be answered. Feel free to uncomment it and give it a shot.

In [90]:
```
# test new user
#user_survey()
```

...

Troubleshooting

trying to figure out why the movie's being reccomended are not within the range. An error was found that was preventing this filter from working the cells below will run now, but there is no need or benefit in running them unless the year filter stops working.

In [91]:
```
#cm = questionnaire()
```

In [92]:
```
#cm
```

In [93]:
```
#recs = n_movies_rec(2,best_svd,cm,5)
#recs
```

It is now working as intended, but these cells can be used to test any code changes to the questionaire function.

Limitations

Our model will only be as good as our data. We should anticipate and expect improved metrics as our data increases.

Recommendations and Next Steps

- Develop plan to encourage users to rate more movies.
- Create new models using user login demographics.
- Use Natural Language Processing(NLP) on movie scripts to gain more insight on our users preferences.

Conclusion

After a thorough iterative modeling process, we determined that the hypertuned svd model would be the best selection at this point. Reccomendations would be made for the new trial users based on their initial reccomendation and the optional survey that limits the years of release and the genres of the recommendations.