Homework 2

In [ ]:
```
"""
Project members:
Weixi Yao (wy2350)
Kushal Wijesundara (kcw2144)
James Ding (jd3703)
Ryan McNally (rom2109)
"""
```

In [7]:
```
import os
import time
import sklearn
import numpy as np
import pandas as pd
import surprise
import seaborn as sns
import matplotlib.pyplot as plt

from utility import *
from scipy import stats
from surprise import NMF
from surprise import Reader
from surprise import Dataset
from surprise import KNNWithMeans
from collections import defaultdict, Counter
from surprise.model_selection import GridSearchCV
from surprise.model_selection import cross_validate
from surprise.model_selection import train_test_split
%matplotlib inline
```

Part 1: Business Objectives

1. Our objective

- The objective of this recommendation system is to increase users' experience/stickiness and ultimately their time spent on our site by making their search for the movies easier as well as by making them consistenly interested in our movie inventory. To achieve that, the recommendation system will provide a list of n (customizable, default 5 and we use 1 for this case study) movies to the users every time they log in, and the recommendation list will be a combination of movies that are most relevant to the users as well as the movies that are novel to the users.

2. Other metrics we wish to optimize, in addition to accuracy

- To echo back to the objective of our recommendation system, we will use item-coverage and novelty as the two metrics we optimize in addition to accuracy.
- Therefore, we will simultaneously optimize for coverage and novelty, the hypothesis being that a sufficiently diverse and new set of recommendations *with* a high accuracy will lead to deeper engagement, more time on the site, more clicks through the recommender, and will yield high quality data for improving the model in later stages.

3. The intended user

- The intended user for the business is any movie watcher & content consumer. Our objective is to maximize the engagement and personalization of this fundamental user. As a core piece of the user experience uplift strategy, this recommendation system intends to serve all users coming to our site to search for movies to watch, which means this recommendation system should be able to make recommendations to both new users and existing users. New users with little to no engagement data will be recommended very diverse and popular movies while their behavior and tastes become evident to the model over time.

4. Business rules we think will be important

- Never recommend items with less than 20% popularity. Our overall strategy will be to "discard" the movies that are the least popular, "maintain" the movies that are the most popular, and "boost" the movies that are in the middle of the popularity spectrum.
- Always include the most popular movies in the recommendation to users who are at the bottom 25% in terms of activeness(measured by the number of movies rated). This will serve as a baseline cold-start strategy.
- Always recommend movies that the users have started but haven't finished if the relevant user data is accessible.
- Always recommend movies that are trending now. For example, recommend romance movies during Valentine's Day.
- Always include some level of novelty in the recommendations, meaning movies from different genres.

5. Performance requirements

- The time complexity of rating making should be O(kN), where N denotes the number of movie items.

6. Notable sacrifices

- As it pertains to the **final recommendation:** There is room for future improvement in testing the recommendation of unpopular movies on subsets of users, and then boosting them pending positive feedback. As it stands, we require a critical mass of ratings for a film to be considered for recommendation by the model (20th percentile of ratings received). A new movie that is never recommended has no hope of getting to that critical mass.
- As it pertains to **optimazation and accuracy:** We are willing to sacrifice maximal 'accuracy' for the deeper and broader goal of attaining a holistic understanding of our users and their tastes. A maximally accurate but non-novel or high-coverage model might converge and recommend the user's favorite movie back to them n times repeatedly; the chances of a user remaining on the site or clicking through over and over in that situation are low.
- As it pertains to **data sampling for model building:** Our sampling method biases for existing popularity, this will ignore currently unpopular users and movies (<50th percentile in the dataset). This is mainly for ensuring richness of signal in the final model which will be recommending *to* sparse datapoint users. This is a business decision, we would rather have a functional beta model to begin with than to erroneously recommend movies with a low network effect, at least for an inital v1. We cannot let the perfect be

the enemy of the good. (We delve into tuning this parameter in Part 6 with impacts on training performance and model performance.)

Part 1.1: Exploratory Data Analysis

Based on the instructions, we downloaded the entire movie data set at https://grouplens.org/datasets/movielens/latest/ with 27M ratings for exploratory data analysis. Within the movie data, the ratings data set consists of 'userids', 'movieids', 'ratings' and 'timestamp' along with the the movies dataset, which consists of 'movieids', 'titles' and 'genres'. For further subsetting and analysis, we converted the 'timestamp' column into year.

In [8]:
```
def rating_movie_df():
    ratings = pd.read_csv('./ml-latest/ratings.csv')
    movies = pd.read_csv('./ml-latest/movies.csv')
    return ratings,movies
```

In [20]:
```
ratings, movies = rating_movie_df()
```

In [21]:
```
# merge data
movie_ratings = ratings.merge(movies, on = 'movieId', how = 'inner')
movie_ratings['timestamp'] = pd.to_datetime(movie_ratings['timestamp'], unit=
movie_ratings.rename(columns = {'timestamp':'year'}, inplace = True)
movie_ratings.head()
```

Out[21]:

	userId	movieId	rating	year	title	genres
0	1	307	3.5	2009	Three Colors: Blue (Trois couleurs: Bleu) (1993)	Drama
1	6	307	4.0	1996	Three Colors: Blue (Trois couleurs: Bleu) (1993)	Drama
2	56	307	4.0	2013	Three Colors: Blue (Trois couleurs: Bleu) (1993)	Drama
3	71	307	5.0	2009	Three Colors: Blue (Trois couleurs: Bleu) (1993)	Drama
4	84	307	3.0	2001	Three Colors: Blue (Trois couleurs: Bleu) (1993)	Drama

In [22]:
```
# numerical data statistics
movie_ratings.describe(include='number').transpose()
```

Out[22]:

	count	mean	std	min	25%	50%	75%	m
userId	27753444.0	141942.015571	81707.400091	1.0	71176.0	142022.0	212459.0	28322
movieId	27753444.0	18487.999834	35102.625247	1.0	1097.0	2716.0	7150.0	19388
rating	27753444.0	3.530445	1.066353	0.5	3.0	3.5	4.0	
year	27753444.0	2007.302610	6.871880	1995.0	2001.0	2007.0	2015.0	201

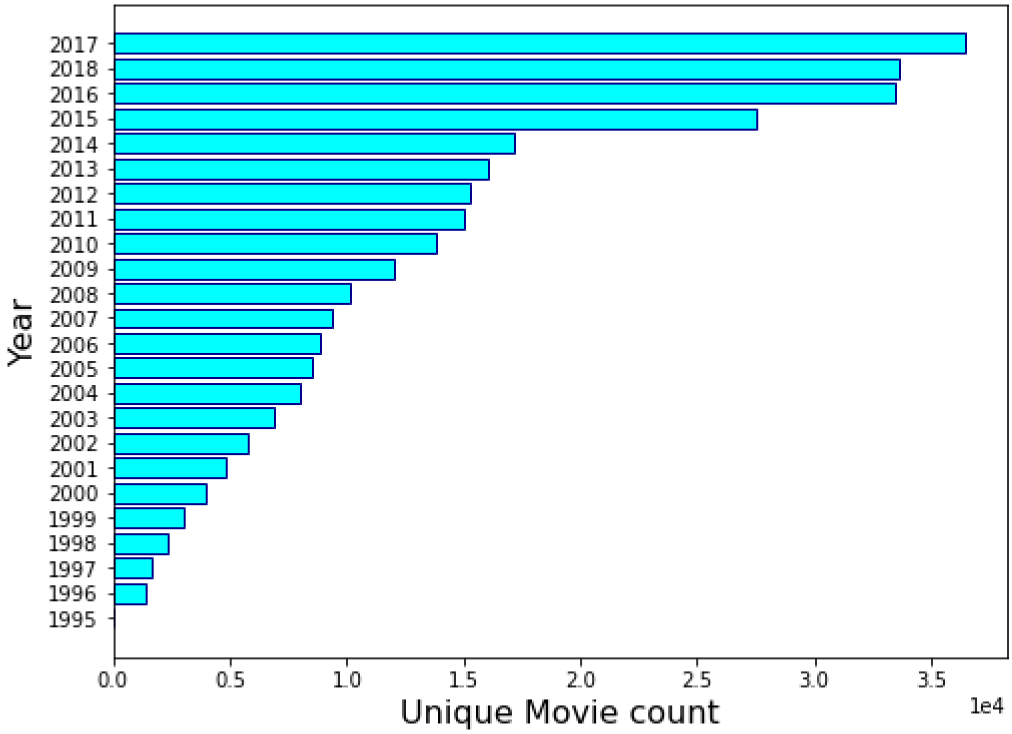
In [23]:
```
#categorical data statistics
movie_ratings.describe(include='object').transpose()
```

Out[23]:

	count	unique	top	freq
title	27753444	53817	Shawshank Redemption, The (1994)	97999
genres	27753444	1610	Drama	1959338

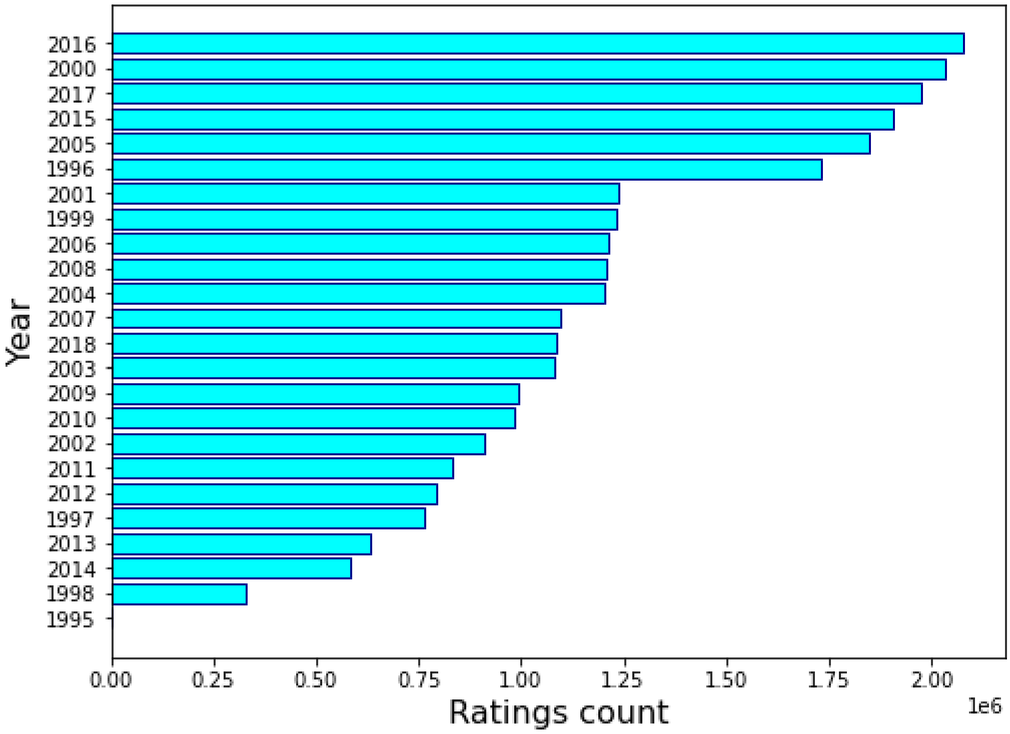
In the following sections, the exploratory data analysis on the Full 27M dataset preset the significance of movies, ratings, and users counts as well as the distribution of ratings, top popular movies, and popular movie distribution.

In [24]:
```
plt.figure(figsize = (8,6))
movie_year = movie_ratings.groupby('year').nunique()[['movieId']].reset_index
year = np.arange(len(movie_year['year']))
plt.barh(year, movie_year['movieId'], align='center', color ='cyan', edgecolo
plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))
#plt.gca().invert_xaxis()
plt.yticks(year, movie_year['year'])
plt.ylabel('Year', fontsize = 16)
plt.xlabel('Unique Movie count', fontsize = 16)
plt.show()
```



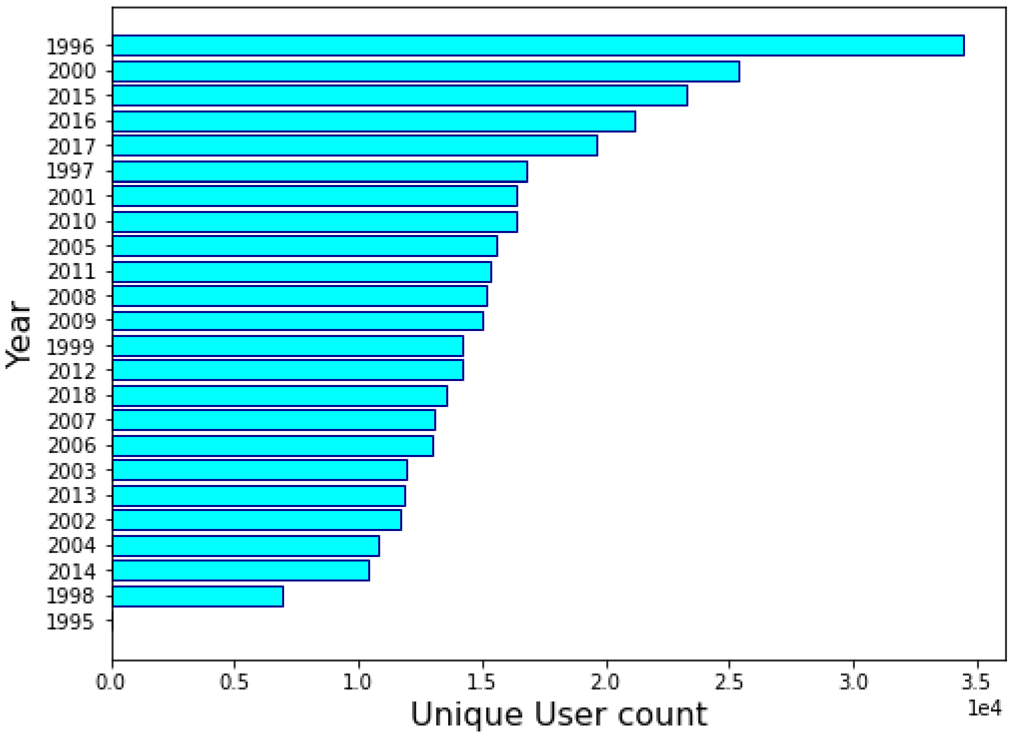
The number of unique movies against the year is presented above. It seems to be that few unique movies in the 90's and it gradually picked up in the 2000's. What is more significant is that the rate almost doubled from 2015 to 2018 compared to the early 2000's. (note: unique movie count in millions)

In [25]:
```
plt.figure(figsize = (8,6))
rating_year = movie_ratings.groupby('year').count()[['rating']].reset_index()
year = np.arange(len(rating_year['year']))
plt.barh(year, rating_year['rating'], align='center', color ='cyan', edgecolo
plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))
#plt.gca().invert_xaxis()
plt.yticks(year, rating_year['year'])
plt.ylabel('Year', fontsize = 16)
plt.xlabel('Ratings count', fontsize = 16)
plt.show()
```



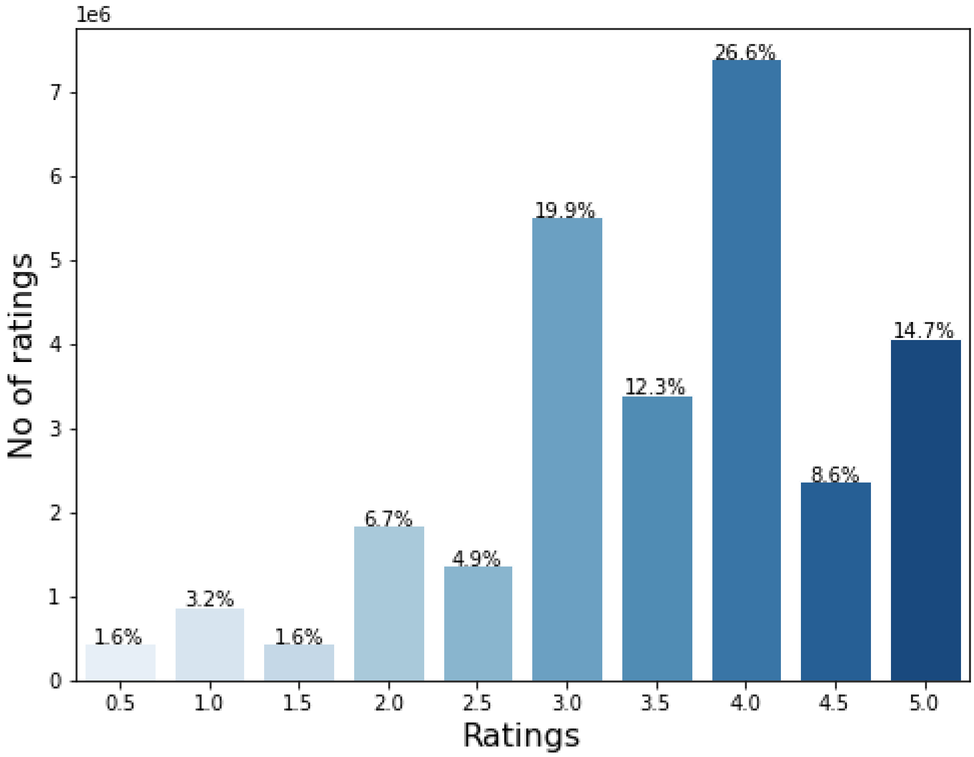
Based on the above histograms, we had the most ratings from 2015 to 2017 were live-streaming started to become increasingly popular, 2005 when mass-streaming started, and the years 1996 and 2000 due to the release of popular movies. (note: ratings count in millions)

In [26]:
```
plt.figure(figsize = (8,6))
user_year = movie_ratings.groupby('year').nunique()[['userId']].reset_index()
year = np.arange(len(user_year['year']))
plt.barh(year, user_year['userId'], align='center', color ='cyan', edgecolor =
plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))
#plt.gca().invert_xaxis()
plt.yticks(year, user_year['year'])
plt.ylabel('Year', fontsize = 16)
plt.xlabel('Unique User count', fontsize = 16)
plt.show()
```



Based on the histograms year 1996 had the most user count while 1995 had the lowest. Further analysis revealed (see last plot for popular movie analysis) most of the popular movies were dated back to 1994, which could account for this behavior. (note: unique user count in millions)

In [27]:
```
plt.figure(figsize = (8,6))
ax = sns.countplot(data = movie_ratings,x='rating', palette = 'Blues', edgeco
ax.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
plt.xlabel('Ratings', fontsize = 16)
plt.ylabel('No of ratings', fontsize = 16)
total = float(len(movie_ratings))
for p in ax.patches:
    height = p.get_height()
    ax.text(p.get_x() + p.get_width()/2.,
            height + 5,
            '{:1.1f}%'.format(100*height/total),
            ha = 'center')
plt.show()
```



The distribution of the ratings for the entire data set is shown above. It presents a negatively skewed distribution where the mean rating and median is above 3. (note: # of ratings in millions)

Part 2: Two models building

In [14]:
```
# load the data set for full ratings
ratings = pd.read_csv('./ml-latest/ratings.csv',
                      usecols=['userId', 'movieId', 'rating'],
                      dtype={'user|Id':'int32', 'movieId':'int32', 'rating':'
                      sep=',')
movies = pd.read_csv('./ml-latest/movies.csv',
                    dtype={'movieId':'int32'})
print("number of distinct participated users: ", ratings['userId'].nunique())
print("number of distinct rated movies: ", ratings['movieId'].nunique())
```

```
number of distinct participated users:  283228
number of distinct rated movies:  53889
```

For experiment and development purposes, we will extract a small subset of users and movie items from the "full" dataset.

We will first select ~1000 items and ~25000 users for model development and we will then select ~200 items and ~1500 users from the development set for testing purposes.

The methodology of selecting these users and items will be discussed in the data exploration section.

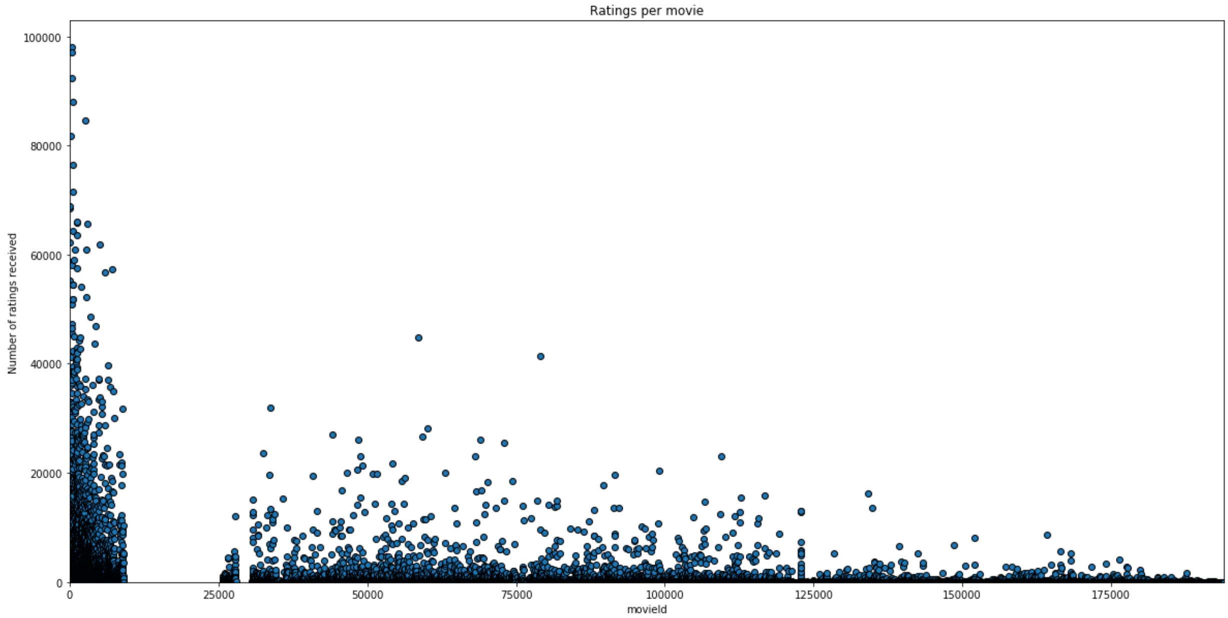
Data exploration and dataset sampling

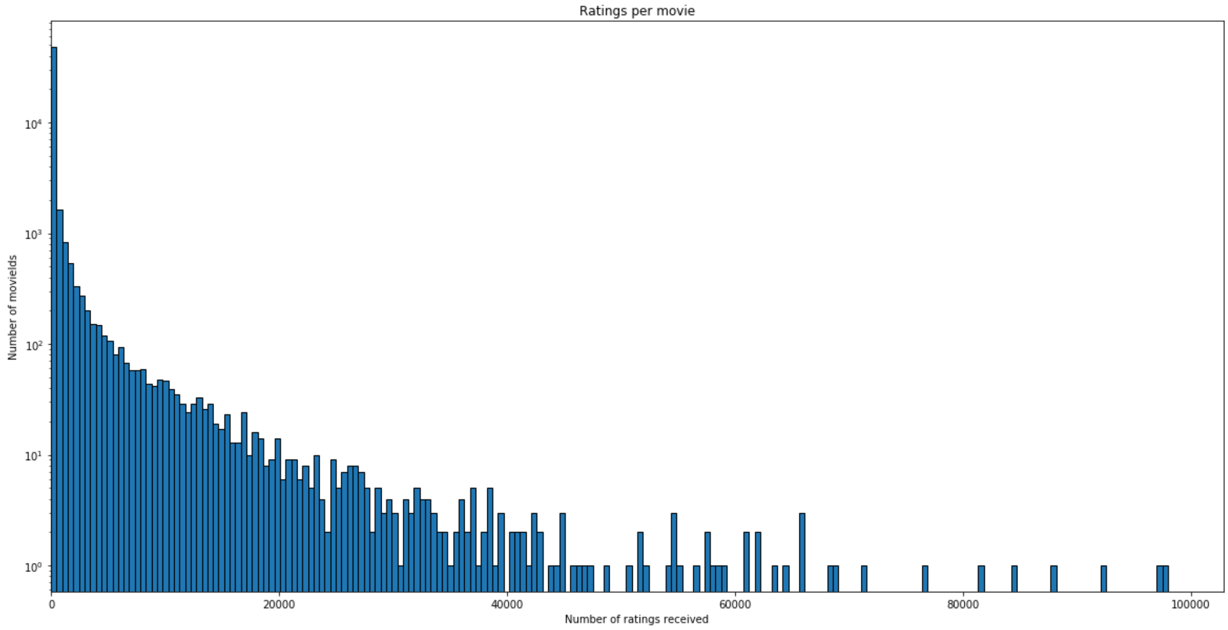
Understanding how the data is distributed is crucial for designing the subsampling strategy and we believe that, in a real world setting, data collected from explicit feedbacks like ratings can be very sparse and data points are mostly collected from very popular movies and highly active users.

The following data exploration will justify our hypothesis.

In [5]:
```
# visualize the popularity by each movie
movie_popularity = ratings[['userId', 'movieId']].groupby('movieId').count()
movie_popularity.columns=['num_ratings']
plt.figure(figsize=(20, 10))
plt.scatter(movie_popularity.index,
            movie_popularity.num_ratings,
            edgecolor='black')
plt.xlim(0, movie_popularity.index.max())
plt.ylim(0,)
plt.title('Ratings per movie')
plt.xlabel('movieId')
plt.ylabel('Number of ratings received')
plt.show()

# visualize the distribution of the popularity across all movies
plt.figure(figsize=(20, 10))
plt.hist(movie_popularity.num_ratings,
         bins=200,
         edgecolor='black',
         log=True)
plt.title('Ratings per movie')
plt.xlabel('Number of ratings received')
plt.ylabel('Number of movieIds')
plt.xlim(0,)
plt.show()
```

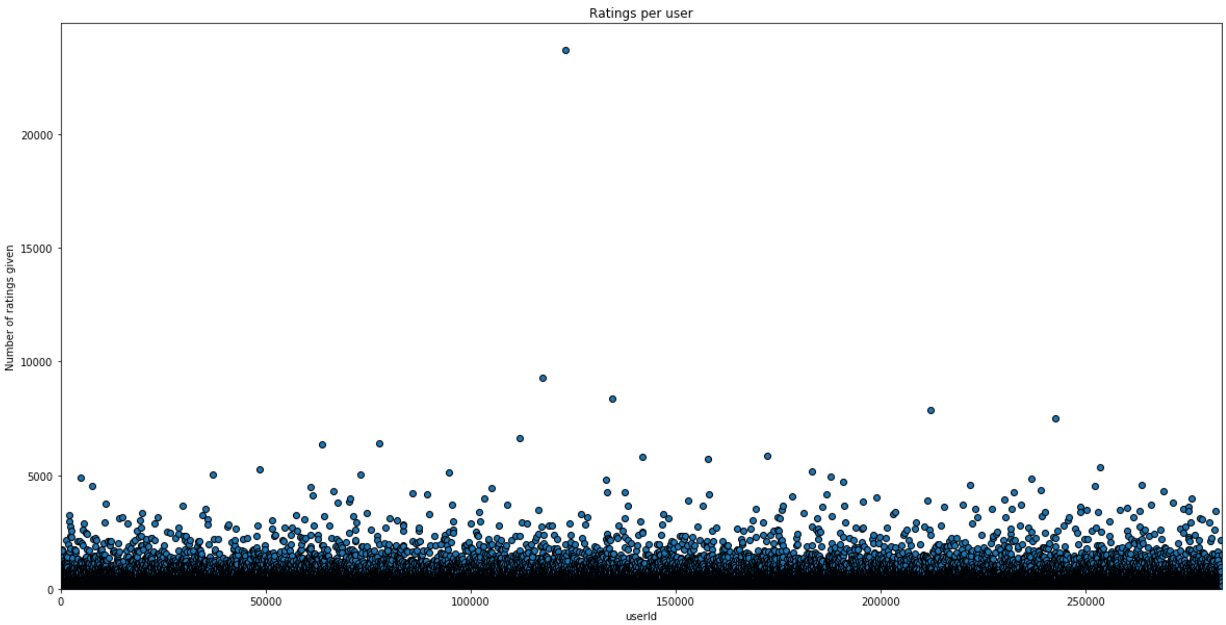


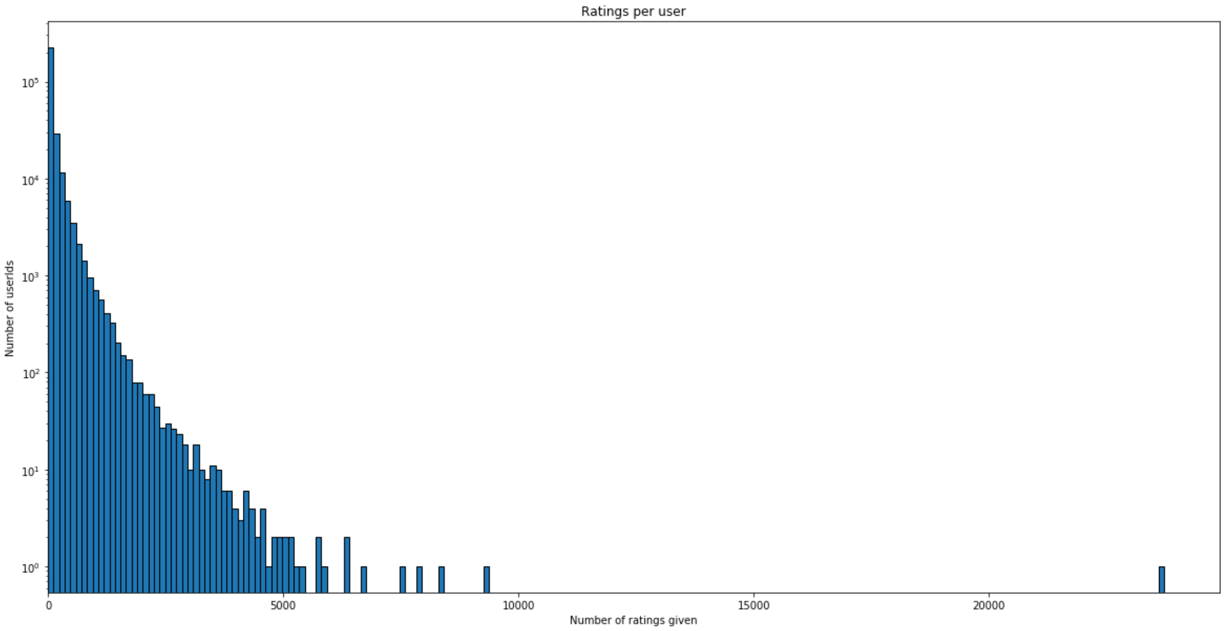


In [6]:
```
# visualize the popularity by each movie
user_activeness = ratings[['userId', 'movieId']].groupby('userId').count()
user_activeness.columns=['num_ratings']
plt.figure(figsize=(20, 10))
plt.scatter(user_activeness.index,
            user_activeness.num_ratings,
            edgecolor='black')
plt.xlim(0, user_activeness.index.max())
plt.ylim(0,)
plt.title('Ratings per user')
plt.xlabel('userId')
plt.ylabel('Number of ratings given')
plt.show()

# visualize the distribution of the popularity across all movies
plt.figure(figsize=(20, 10))
plt.hist(user_activeness.num_ratings,
         bins=200,
         edgecolor='black',
         log=True)
plt.title('Ratings per user')
plt.xlabel('Number of ratings given')
plt.ylabel('Number of userIds')
```

```
plt.xlim(0,)
plt.show()
```





- The graphs above preliminarily confirm our assumptions regarding the distribution of the number of ratings received per movie and the distribution of the number of ratings given per user with a few particularly enthusiastic viewers, which are two extremely long-tailed distributions. Based on this observation, we ideally want to draw a subset of movie items and users that can represent the original dataset without giving unrealistically good or bad experiment results.

- We specifically want to get rid of the movies whose popularity is at the bottom 50 percentile, because we believe that there is a high possibility that those movies are only favored by an extremely small group of users, and more importantly, that the average ratings received by those movies are more likely to be biased. The same strategy also applies to users.

- For example, if certain movies are rated only once and received an average rating of 5, in an item-based collaborative filtering model, those movies will accidentally become part of the recommendations for many if not all users, which is not ideal.

Create a subset of dataset, save it for performance measurement

In [7]:
```
ratings_full = sample_dateset_by_percentile(ratings, 25000, 1000)
print("number of distinct participated users (full): ", ratings_full['userId'
print("number of distinct rated movies (full): ", ratings_full['movieId'].nun
```

```
number of distinct participated users (full):  24511
number of distinct rated movies (full):  1000
```

In [8]:
```
reader = Reader()
data_full = Dataset.load_from_df(ratings_full[['userId',
                                               'movieId',
                                               'rating']],
                                 reader=reader)
```

Create a subset of dataset, save it for development

In [9]:
```
ratings_sub = sample_dateset_by_percentile(ratings_full, 1500, 200)
print("number of distinct participated users (subset): ", ratings_sub['userId
print("number of distinct rated movies (subset): ", ratings_sub['movieId'].nu
```

```
number of distinct participated users (subset):  1487
number of distinct rated movies (subset):  200
```

In [10]:
```
reader = Reader()
data_sub = Dataset.load_from_df(ratings_sub[['userId',
                                             'movieId',
                                             'rating']],
                                reader=reader)
```

After careful manipulation, we have successfully keep 1487 users from top active levels and 200 movie items from top popularity levels, for development purposes.

Part 2.1: Item-based neighborhood method

In [11]:
```
trainvalset, testset = train_test_split(data_sub, test_size=0.2)
```

In [12]:
```
# baseline modeling constructing
item_model = KNNWithMeans(k=4, sim_options={'name': 'pearson',
                                            'user_based': False,
                                            'verbose' : False})
```

In [13]:
```
item_model.fit(trainvalset);
```

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

In [14]:
```
uid = str(699)
iid = str(296)
item_model_pred = item_model.predict(uid,
                                     iid,
                                     verbose=False)
print("Prediction for rating: ", item_model_pred)
```

```
Prediction for rating:  user: 699        item: 296        r_ui = None   est =
3.50   {'was_impossible': True, 'reason': 'User and/or item is unknown.'}
```

- In this project, we'll use the surprise package, a popular package for building recommendation systems in Python.
- At this part, we have sucessfully built and tested a baseline item-based neighborhood model by making a demo rating pradiction for user 699 and item 296.

- The item-based neighborhood model built above uses pearson as the similarity metric and considers the ratings of four closest neighbors while making predictions.

Part 2.2: Matrix Factorization

In [19]:
```
# baseline model constructing
NMF_model = NMF(n_factors=20,
                n_epochs=10,
                biased=True)
```

In [20]:
```
NMF_model.fit(trainvalset);
```

In [21]:
```
uid = str(699)
iid = str(296)
NMF_model_pred = NMF_model.predict(uid,
                                   iid,
                                   verbose=False)
print("Prediction for rating: ", NMF_model_pred)
```

```
Prediction for rating:  user: 699        item: 296        r_ui = None   est =
3.50   {'was_impossible': False}
```

- At this part, we have sucessfully built and tested a baseline matrix factorization model(NMF) by making a demo rating pradiction for user 699 and item 296.
- The NMF model built above decomposes the characteristic matrix in latent space size of 20.

Part 3: Model evaluation

Part 3.1.1: CV setup for item-based neighborhood method

For this task, we will use **RMSE(Root Mean Squared Error)** as the primary accuracy metric and **MAE(Mean Absolute Error)** as the secondary accuracy metric.

In [22]:
```
# Run 5-fold cross-validation and print results
result = cross_validate(item_model,
                        data_full,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
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)    0.9485  0.9455  0.9549  0.9515  0.9470  0.9495  0.0034
MAE (testset)     0.7174  0.7168  0.7216  0.7168  0.7168  0.7179  0.0019
RMSE (trainset)   0.4178  0.4177  0.4184  0.4184  0.4192  0.4183  0.0005
MAE (trainset)    0.3122  0.3119  0.3128  0.3126  0.3134  0.3126  0.0005
```

```
Fit time          0.87    0.91    0.89    0.88    0.89    0.89    0.01
Test time         1.93    2.10    1.89    1.90    1.90    1.95    0.08
```

The results above give an preliminary review of how well the model fits the training set as well as how well the model generalizes on the test set. According to the results above, the **average testset RMSE is 0.95** and the **average testset MAE is 0.72**.

Part 3.1.2: Coverage of item-based neighborhood method

In [23]:
```
trainvalset, testset = train_test_split(data_sub, test_size=0.2)
trainvalset_testfy = trainvalset.build_anti_testset()
```

In [24]:
```
user_list_train = set(trainvalset.all_users())
user_list_test = set([item[0] for item in testset])

item_list_train = set(trainvalset.all_items())
item_list_test = set([item[1] for item in testset])
```

In [25]:
```
item_model.fit(trainvalset);
```

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

In [26]:
```
predictions_train = item_model.test(trainvalset_testfy)
predictions_test = item_model.test(testset)

top_n_train = get_top_n(predictions_train, n=1)
top_n_test = get_top_n(predictions_test, n=1)
```

In [27]:
```
recommendation_train = []
# Print the recommended items for each user
for uid, user_ratings in top_n_train.items():
    recommendation_train.append([iid for (iid, _) in user_ratings])

recommendation_train_list = []
for rec_list in recommendation_train:
    for item in rec_list:
        recommendation_train_list.append(item)
recommendation_train_list = set(recommendation_train_list)
```

In [28]:
```
recommendation_test = []
# Print the recommended items for each user
for uid, user_ratings in top_n_test.items():
    recommendation_test.append([iid for (iid, _) in user_ratings])

recommendation_test_list = []
for rec_list in recommendation_test:
    for item in rec_list:
        recommendation_test_list.append(item)
recommendation_test_list = set(recommendation_test_list)
```

In [29]:
```
print('coverage on train set', '{:.2f}'.format(len(recommendation_train_list)
print('coverage on test set', '{:.2f}'.format(len(recommendation_test_list)/l
```

```
coverage on train set 0.29
coverage on test set 0.67
```

- As we can see with the above discrepancy in coverage between the train and test sets, our recommender is converging on a few top performers in the train set but is more dispersed in the test set. We will test this hypothesis below by checking how often, on a percentage basis, the top movies were recommended in each case.

- With the lower coverage, we would expect that the top recommendations are recommended more often in the train set.

In [77]:
```
train_counter = pd.DataFrame(Counter([item for sublist in  recommendation_tra
train_counter['pct_recommended'] = train_counter['count']/len(recommendation_
test_counter = pd.DataFrame(Counter([item for sublist in  recommendation_test
test_counter['pct_recommended'] = test_counter['count']/len(recommendation_te
```

In [80]:
```
train_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[80]:

	title	count	pct_recommended
0	Toy Story (1995)	418	0.286301
1	Jumanji (1995)	270	0.184932
2	Grumpier Old Men (1995)	155	0.106164
3	Waiting to Exhale (1995)	87	0.059589
4	Father of the Bride Part II (1995)	86	0.058904

In [81]:
```
test_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[81]:

	title	count	pct_recommended
0	Toy Story (1995)	186	0.198930
1	Jumanji (1995)	58	0.062032
2	Grumpier Old Men (1995)	37	0.039572
3	Waiting to Exhale (1995)	36	0.038503
4	Father of the Bride Part II (1995)	29	0.031016

- Our hypothesis is confirmed, we see that the test case recommended the most popular title (Toy Story) 19.89% of the time , while the train set was recommended it over 28.6% of the time.
- The discrepancy is driven by the 'richer rich' effect of the top performers nabbing the n=1 top spot.
- The next hypothesis will be to increase n to 5 and we will ideally see higher coverage in both sets, and *relative* closer parity between coverages.

In [82]:
```
predictions_train = item_model.test(trainvalset_testfy)
predictions_test = item_model.test(testset)

#new hypothesis n=5
top_n_train = get_top_n(predictions_train, n=5)
top_n_test = get_top_n(predictions_test, n=5)

recommendation_train = []
# Print the recommended items for each user
for uid, user_ratings in top_n_train.items():
    recommendation_train.append([iid for (iid, _) in user_ratings])

recommendation_train_list = []
for rec_list in recommendation_train:
    for item in rec_list:
        recommendation_train_list.append(item)
recommendation_train_list = set(recommendation_train_list)
```

```
recommendation_test = []
# Print the recommended items for each user
for uid, user_ratings in top_n_test.items():
    recommendation_test.append([iid for (iid, _) in user_ratings])

recommendation_test_list = []
for rec_list in recommendation_test:
    for item in rec_list:
        recommendation_test_list.append(item)
recommendation_test_list = set(recommendation_test_list)

print('coverage on train set', '{:.2f}'.format(len(recommendation_train_list)
print('coverage on test set', '{:.2f}'.format(len(recommendation_test_list)/l
```

```
coverage on train set 0.65
coverage on test set 0.97
```

In [83]:
```
train_counter = pd.DataFrame(Counter([item for sublist in  recommendation_tra
train_counter['pct_recommended'] = train_counter['count']/len(recommendation_
test_counter = pd.DataFrame(Counter([item for sublist in  recommendation_test
test_counter['pct_recommended'] = test_counter['count']/len(recommendation_te
```

In [84]:
```
train_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[84]:

	title	count	pct_recommended
0	Toy Story (1995)	769	0.526712
1	Jumanji (1995)	767	0.525342
2	Grumpier Old Men (1995)	669	0.458219
3	Waiting to Exhale (1995)	592	0.405479
4	Father of the Bride Part II (1995)	454	0.310959

In [85]:
```
test_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[85]:

	title	count	pct_recommended
0	Toy Story (1995)	213	0.227807
1	Jumanji (1995)	81	0.086631
2	Grumpier Old Men (1995)	79	0.084492
3	Waiting to Exhale (1995)	65	0.069519
4	Father of the Bride Part II (1995)	59	0.063102

Explanation:

- The disparity between train and test speaks to potential overfitting of this model, higher coverage is desirable but is likely being seen in this case because the model is much more unsure of what it is meant to recommend on unseen users.
- While there is potentially a degree of overfitting, we are encouraged by the fact that the same top performaers are recommended in each case and in the same order, just with varying frequencies.
- The best remedy for this is to either regularize the model and penalize convergence in the train set more, but this is sub-optimal in practice. We would much rather see more

user data to enrich the model further by adding more features, rather than penalize the model for high bias low variance.

Part 3.2.1: CV setup for Matrix Factorization

In [82]:
```
# Run 5-fold cross-validation and print results
result = cross_validate(NMF_model,
                        data_full,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.1466  1.1935  1.2727  1.1738  1.1380  1.1849  0.0481
MAE (testset)     0.8672  0.9039  0.9661  0.8884  0.8603  0.8972  0.0378
RMSE (trainset)   1.1337  1.1863  1.2443  1.1647  1.1252  1.1708  0.0427
MAE (trainset)    0.8506  0.8934  0.9413  0.8767  0.8494  0.8823  0.0338
Fit time          3.66    3.72    3.50    3.45    3.49    3.56    0.11
Test time         0.41    0.42    0.38    0.96    1.02    0.64    0.29
```

The results above give an preliminary review of how well the model fits the training set as well as how well the model generalizes on the test set. According to the results above, the **average testset RMSE is 1.185** and the **average testset MAE is 0.897**.

Part 3.2.2: Coverage of Matrix Factorization

In [86]:
```
NMF_model.fit(trainvalset);
```

In [87]:
```
predictions_train = NMF_model.test(trainvalset_testfy)
predictions_test = NMF_model.test(testset)

top_n_train = get_top_n(predictions_train, n=1)
top_n_test = get_top_n(predictions_test, n=1)
```

In [88]:
```
recommendation_train = []
# Print the recommended items for each user
for uid, user_ratings in top_n_train.items():
    recommendation_train.append([iid for (iid, _) in user_ratings])

recommendation_train_list = []
for rec_list in recommendation_train:
    for item in rec_list:
        recommendation_train_list.append(item)
recommendation_train_list = set(recommendation_train_list)
```

In [89]:
```
recommendation_test = []
# Print the recommended items for each user
for uid, user_ratings in top_n_test.items():
    recommendation_test.append([iid for (iid, _) in user_ratings])

recommendation_test_list = []
for rec_list in recommendation_test:
    for item in rec_list:
        recommendation_test_list.append(item)
recommendation_test_list = set(recommendation_test_list)
```

In [90]:
```
print('coverage on train set', '{:.2f}'.format(len(recommendation_train_list)
print('coverage on test set', '{:.2f}'.format(len(recommendation_test_list)/le
```

```
coverage on train set 0.13
coverage on test set 0.74
```

In [91]:

```
train_counter = pd.DataFrame(Counter([item for sublist in  recommendation_tra
train_counter['pct_recommended'] = train_counter['count']/len(recommendation_
test_counter = pd.DataFrame(Counter([item for sublist in  recommendation_test
test_counter['pct_recommended'] = test_counter['count']/len(recommendation_te
```

In [92]:

```
train_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[92]:

	title	count	pct_recommended
0	Toy Story (1995)	1097	0.751370
1	Jumanji (1995)	125	0.085616
2	Grumpier Old Men (1995)	42	0.028767
3	Waiting to Exhale (1995)	30	0.020548
4	Father of the Bride Part II (1995)	29	0.019863

In [93]:

```
test_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[93]:

	title	count	pct_recommended
0	Toy Story (1995)	189	0.202139
1	Jumanji (1995)	61	0.065241
2	Grumpier Old Men (1995)	39	0.041711
3	Waiting to Exhale (1995)	36	0.038503
4	Father of the Bride Part II (1995)	29	0.031016

- Here we see we have much the same effect as with matrix factorization as we do with kNN. Higher coverage on test than train, and concentration among top performers in train.
- Below we run the same experiment, increasing n to 5:

In [95]:

```
top_n_train = get_top_n(predictions_train, n=5)
top_n_test = get_top_n(predictions_test, n=5)

recommendation_train = []
# Print the recommended items for each user
for uid, user_ratings in top_n_train.items():
    recommendation_train.append([iid for (iid, _) in user_ratings])

recommendation_train_list = []
for rec_list in recommendation_train:
    for item in rec_list:
        recommendation_train_list.append(item)
recommendation_train_list = set(recommendation_train_list)

recommendation_test = []
# Print the recommended items for each user
for uid, user_ratings in top_n_test.items():
    recommendation_test.append([iid for (iid, _) in user_ratings])

recommendation_test_list = []
for rec_list in recommendation_test:
```

```
    for item in rec_list:
        recommendation_test_list.append(item)
recommendation_test_list = set(recommendation_test_list)

print('coverage on train set', '{:.2f}'.format(len(recommendation_train_list)
print('coverage on test set', '{:.2f}'.format(len(recommendation_test_list)/l
```

```
coverage on train set 0.26
coverage on test set 0.97
```

In [96]:
```
train_counter = pd.DataFrame(Counter([item for sublist in  recommendation_tra
train_counter['pct_recommended'] = train_counter['count']/len(recommendation_
test_counter = pd.DataFrame(Counter([item for sublist in  recommendation_test
test_counter['pct_recommended'] = test_counter['count']/len(recommendation_te
```

In [97]:
```
train_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[97]:

	title	count	pct_recommended
0	Toy Story (1995)	1356	0.928767
1	Jumanji (1995)	1299	0.889726
2	Grumpier Old Men (1995)	816	0.558904
3	Waiting to Exhale (1995)	787	0.539041
4	Father of the Bride Part II (1995)	757	0.518493

In [98]:
```
test_counter.join(movies).head()[['title','count','pct_recommended']]
```

Out[98]:

	title	count	pct_recommended
0	Toy Story (1995)	214	0.228877
1	Jumanji (1995)	81	0.086631
2	Grumpier Old Men (1995)	81	0.086631
3	Waiting to Exhale (1995)	64	0.068449
4	Father of the Bride Part II (1995)	58	0.062032

Explanation:

- As with kNN, the explanation here is that the model has higher likelihood of recommending a random movie in the test dataset than the train dataset.

Part 4: GridSearch

Part 4.1: GS setup for item-based neighborhood method

In [99]:
```
param_grid = {'k': [2, 6, 10, 14, 18, 20, 50, 100]}
```

In [100…
```
gs = GridSearchCV(KNNWithMeans,
                  param_grid,
                  measures=['RMSE', 'MAE'],
                  cv=5)
```

In [101…
```
gs.fit(data_sub);
```

```
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.
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.
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.
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.
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.
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.
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.
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.
```

In [102…

```
results_df = pd.DataFrame.from_dict(gs.cv_results)
```

In [103…

```
n_k = results_df['param_k']
mae_results = results_df['mean_test_mae']
rmse_results = results_df['mean_test_rmse']
```

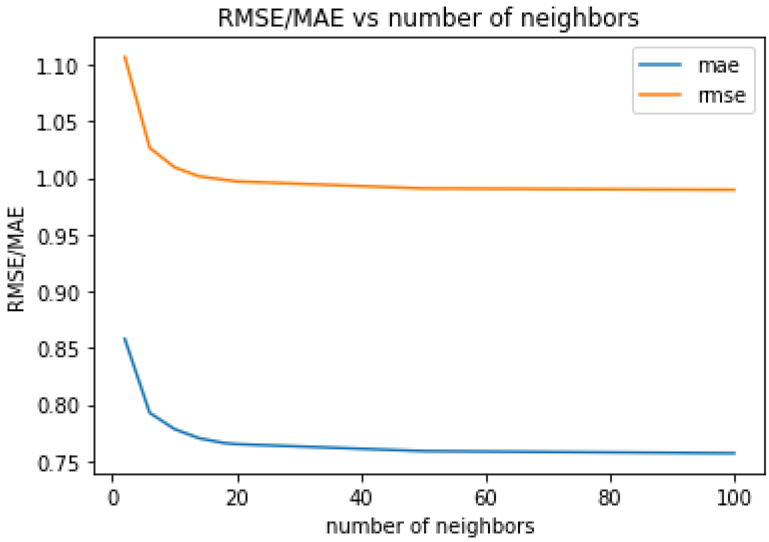
In [104…

```
# best RMSE score
print('Best RMSE', '{:f}'.format(gs.best_score['rmse']))
print('BEST MAE', '{:f}'.format(gs.best_score['mae']))
```

```
Best RMSE 0.989426
BEST MAE 0.756856
```

In [105…

```
fig, ax = plt.subplots()
ax.plot(n_k, mae_results, label='mae');
ax.plot(n_k, rmse_results, label='rmse');
ax.set_title('RMSE/MAE vs number of neighbors')
ax.set_xlabel('number of neighbors')
ax.set_ylabel('RMSE/MAE')
ax.legend();
```



Explanation:

- As we can see from the results above, as the number of neighbors increases, both of the accuracy measures decrease. This makes sense because as the model considers more items(neighbors) while making predictions, the model becomes more capable of making more accurate predictions.
- Although we haven't reached the optimal RMSE/MAE in this grid, we would say the **best number of neighbors is roughly 15**:
  - We come to the above conclusion taking performance into account, there is a diminishing return on computation power vs incremental improvement in the error term. Depending on the frequency this model will refresh against the amount of

time/energy/memory it will take to do so, we think 15 is a reasonable number on these grounds.

- Adding more neighbors is not a cure-all either, too few neighbors and you might miss the 'cluster' you're trying to participate in, too many neighbors and you can shift your mean away from a strong-signal cluster with relatively few peers and this can actually increase error on a case-by-case basis but will smooth out the error over the full model.
- Imagine the qualitative neighbors of a movie like *Rocky Horror Picture Show*... there aren't many. So a modest k is ideal here. Setting it exhorbitantly high will approach a utilitarian model of 'the least bad recommendation for the most people' but offers very little of the personalization we are after here.
- Thus it is normally recommended to pick a k near the inflection point where the error term levels off. We endorse this notion in our proposal.

Part 4.2: GS setup for Matrix Factorization

In [106…

```
param_grid = {'n_factors': [10, 30, 50, 70, 90]}
```

In [107…

```
gs = GridSearchCV(NMF,
                  param_grid,
                  measures=['RMSE', 'MAE'],
                  cv=5)
```

In [108…

```
gs.fit(data_sub)
```

In [110…

```
results_df = pd.DataFrame.from_dict(gs.cv_results)
```

In [111…

```
n_factors = results_df['param_n_factors']
mae_results = results_df['mean_test_mae']
rmse_results = results_df['mean_test_rmse']
```

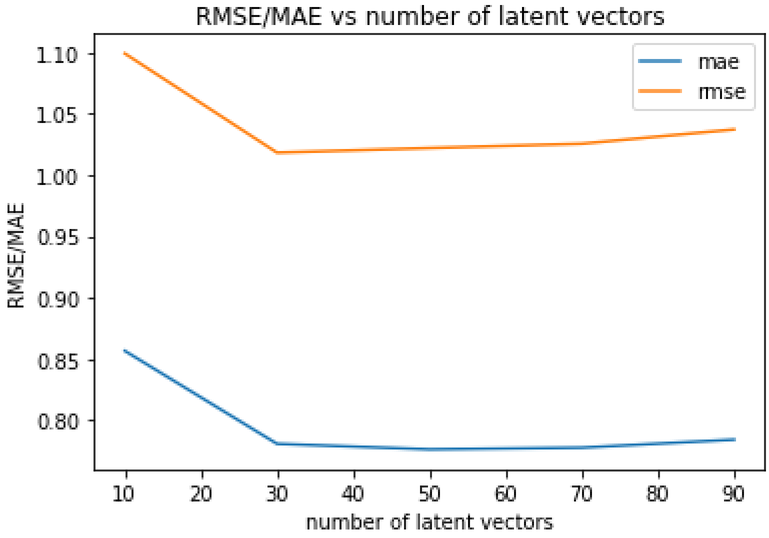
In [112…

```
# best RMSE score
print('Best RMSE', '{:f}'.format(gs.best_score['rmse']))
print('BEST MAE', '{:f}'.format(gs.best_score['mae']))
```

```
Best RMSE 1.018175
BEST MAE 0.776286
```

In [113…

```
fig, ax = plt.subplots()
ax.plot(n_factors, mae_results, label='mae');
ax.plot(n_factors, rmse_results, label='rmse');
ax.set_title('RMSE/MAE vs number of latent vectors')
ax.set_xlabel('number of latent vectors')
ax.set_ylabel('RMSE/MAE')
ax.legend();
```



Explanation:

- As we can see from the results above, as the number of latent vectors increases, both of the accuracy measures decrease. This makes sense because the matrix factorization algorithm re-constructs the true matrix more accurately with higher latent vector size.

- **The best size of latent space is 30**

Part 5: Other design

There are two key observations arise from the above section:

1. Models with higher complexity usually perform better.
    - According to the grid-search results, the "accuracy" of the models increase as the number of latent vectors/number of neighbors increase. Therefore, if we want to further increase the accuracy of the baseline models built previously, we should definitely construct models with reasonably high complexity.

1. The test set's coverage of the more "accurate" model is lower.
    - The tradeoff with complexity is overfitting. We love accuracy and low error, but we are looking to learn holistic functions about our dataset, not memorize it. There is a significant inflection point around the 30 feature latent space range, hence our decision to parameterize our model on that point.

Part 6: Sample size modification

Part 6.1: 25%

In [251…

```
ratings_25 = sample_dateset_by_percentile(ratings, 8000, 250)
print("number of distinct participated users (full): ", ratings_25['userId'].
print("number of distinct rated movies (full): ", ratings_25['movieId'].nuniq
```

```
number of distinct participated users (full):  6370
number of distinct rated movies (full):  250
```

In [252…

```
reader = Reader()
data_25 = Dataset.load_from_df(ratings_25[['userId',
                                            'movieId',
```

```
                                    'rating']],
                             reader=reader)
```

rebuild the model using the best paramter

In [253…

```
# rebuild the model using the best paramter
item_model = KNNWithMeans(k=15,
                          sim_options={'name': 'pearson',
                                       'user_based': False,
                                       'verbose' : False})
```

In [254…

```
# rebuild the model using the best paramter
NMF_model = NMF(n_factors=30,
                n_epochs=10,
                biased=True)
```

Part 6.1.1: Performance of item-based neighborhood method at 25% sampling size

In [255…

```
result = cross_validate(item_model,
                        data_25,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
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)    1.0098  0.9871  1.0034  0.9861  1.0016  0.9976  0.0094
MAE (testset)     0.7628  0.7410  0.7533  0.7461  0.7591  0.7525  0.0081
RMSE (trainset)   0.4358  0.4384  0.4313  0.4353  0.4353  0.4352  0.0023
MAE (trainset)    0.3048  0.3062  0.3011  0.3043  0.3043  0.3041  0.0017
Fit time          0.04    0.05    0.05    0.05    0.05    0.05    0.00
Test time         0.14    0.13    0.13    0.13    0.14    0.13    0.00
```

In [256…

```
item_model_25_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
item_model_25_test_time = np.sum(result['test_time'])/len(result['test_time']
```

In [257…

```
item_model_25_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
item_model_25_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.1.2: Performance of Matrix Factroization at 25% sampling size

In [258…

```
result = cross_validate(NMF_model,
                        data_25,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

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

```
                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.4285  1.4861  1.4817  1.3912  1.3343  1.4244  0.0571
MAE (testset)     1.1089  1.1827  1.1454  1.0807  1.0218  1.1079  0.0551
RMSE (trainset)   1.4141  1.4923  1.4808  1.3792  1.3360  1.4205  0.0595
MAE (trainset)    1.0872  1.1681  1.1423  1.0563  1.0253  1.0958  0.0529
Fit time          0.63    0.62    0.59    0.58    0.57    0.60    0.02
Test time         0.04    0.04    0.04    0.04    0.04    0.04    0.00
```

In [259…

```
NMF_model_25_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
NMF_model_25_test_time = np.sum(result['test_time'])/len(result['test_time'])
```

In [260…

```
NMF_model_25_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
NMF_model_25_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.2: 50%

In [261…

```
ratings_50 = sample_dateset_by_percentile(ratings, 13000, 500)
print("number of distinct participated users (full): ", ratings_50['userId'].
print("number of distinct rated movies (full): ", ratings_50['movieId'].nuniq
```

```
number of distinct participated users (full):  12131
number of distinct rated movies (full):  500
```

In [262…

```
reader = Reader()
data_50 = Dataset.load_from_df(ratings_50[['userId',
                                           'movieId',
                                           'rating']],
                               reader=reader)
```

Part 6.2.1: Performance of item-based neighborhood method at 50% sampling size

In [263…

```
result = cross_validate(item_model,
                        data_50,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
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)    0.9298  0.9294  0.9343  0.9281  0.9417  0.9326  0.0050
MAE (testset)     0.7072  0.7033  0.7084  0.7025  0.7159  0.7075  0.0048
RMSE (trainset)   0.4893  0.4942  0.4944  0.4900  0.4921  0.4920  0.0021
MAE (trainset)    0.3584  0.3620  0.3620  0.3592  0.3608  0.3605  0.0015
Fit time          0.19    0.23    0.22    0.21    0.22    0.22    0.01
Test time         1.23    0.66    0.65    0.63    0.64    0.76    0.23
```

In [264…

```
item_model_50_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
item_model_50_test_time = np.sum(result['test_time'])/len(result['test_time']
```

In [265…

```
item_model_50_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
item_model_50_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.2.2: Performance of Matrix Factorization at 50% sampling size

In [266…
```
result = cross_validate(NMF_model,
                        data_50,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.3238  1.4741  1.2383  1.6707  1.3311  1.4076  0.1518
MAE (testset)     1.0182  1.1585  0.9527  1.3294  1.0153  1.0948  0.1353
RMSE (trainset)   1.3299  1.4370  1.2335  1.6646  1.3225  1.3975  0.1483
MAE (trainset)    1.0179  1.1193  0.9403  1.3255  1.0050  1.0816  0.1348
Fit time          1.63    1.65    1.75    1.76    1.67    1.69    0.05
Test time         0.77    0.14    0.15    0.14    0.14    0.27    0.25
```

In [267…
```
NMF_model_50_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
NMF_model_50_test_time = np.sum(result['test_time'])/len(result['test_time'])
```

In [268…
```
NMF_model_50_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
NMF_model_50_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.3: 75%

In [269…
```
ratings_75 = sample_dateset_by_percentile(ratings, 19000, 750)
print("number of distinct participated users (full): ", ratings_75['userId'].
print("number of distinct rated movies (full): ", ratings_75['movieId'].nuniq
```

```
number of distinct participated users (full):  18377
number of distinct rated movies (full):  750
```

In [270…
```
reader = Reader()
data_75 = Dataset.load_from_df(ratings_75[['userId',
                                           'movieId',
                                           'rating']],
                               reader=reader)
```

Part 6.3.1: Performance of item-based neighborhood method at 75% sampling size

In [271…
```
result = cross_validate(item_model,
                        data_75,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
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)    0.9264  0.9241  0.9281  0.9231  0.9250  0.9253  0.0018
MAE (testset)     0.6964  0.6988  0.6971  0.6940  0.6968  0.6966  0.0015
RMSE (trainset)   0.5150  0.5177  0.5139  0.5159  0.5161  0.5158  0.0013
MAE (trainset)    0.3796  0.3825  0.3792  0.3807  0.3811  0.3806  0.0012
Fit time          0.60    0.49    0.46    0.48    0.50    0.50    0.05
Test time         2.34    1.45    1.48    1.94    1.43    1.73    0.36
```

In [272…

```
item_model_75_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
item_model_75_test_time = np.sum(result['test_time'])/len(result['test_time']
```

In [273…

```
item_model_75_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
item_model_75_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.3.2: Performance of Matrix Factorization at 75% sampling size

In [274…

```
result = cross_validate(NMF_model,
                        data_75,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2147  1.2451  1.3294  1.2712  1.2098  1.2540  0.0437
MAE (testset)     0.9204  0.9561  1.0071  0.9692  0.9175  0.9541  0.0332
RMSE (trainset)   1.1938  1.2219  1.3231  1.2621  1.1918  1.2385  0.0494
MAE (trainset)    0.9015  0.9328  0.9979  0.9550  0.9004  0.9375  0.0365
Fit time          3.21    2.95    2.89    3.01    3.19    3.05    0.13
Test time         0.27    0.86    0.25    0.27    0.28    0.39    0.24
```

In [275…

```
NMF_model_75_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
NMF_model_75_test_time = np.sum(result['test_time'])/len(result['test_time'])
```

In [276…

```
NMF_model_75_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
NMF_model_75_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.4: 100%

In [277…

```
ratings_100 = sample_dateset_by_percentile(ratings, 25000, 1000)
print("number of distinct participated users (full): ", ratings_100['userId']
print("number of distinct rated movies (full): ", ratings_100['movieId'].nuni
```

```
number of distinct participated users (full):  24511
number of distinct rated movies (full):  1000
```

In [278…

```
reader = Reader()
data_100 = Dataset.load_from_df(ratings_100[['userId',
                                             'movieId',
                                             'rating']],
                                reader=reader)
```

Part 6.4.1: Performance of item-based neighborhood method at 100% sampling size

In [279…

```
result = cross_validate(item_model,
                        data_100,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.
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)    0.9122  0.9176  0.9130  0.9169  0.9178  0.9155  0.0024  
MAE (testset)     0.6893  0.6928  0.6914  0.6918  0.6941  0.6919  0.0016  
RMSE (trainset)   0.5211  0.5213  0.5231  0.5190  0.5202  0.5209  0.0014  
MAE (trainset)    0.3870  0.3868  0.3886  0.3855  0.3863  0.3869  0.0010  
Fit time          0.85    0.81    0.76    0.80    0.80    0.80    0.03    
Test time         2.43    2.28    2.27    2.27    2.84    2.42    0.22    
```

In [280…
```
item_model_100_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
item_model_100_test_time = np.sum(result['test_time'])/len(result['test_time'
```

In [281…
```
item_model_100_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
item_model_100_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.4.2: Performance of Matrix Factorization at 100% sampling size

In [282…
```
result = cross_validate(NMF_model,
                        data_100,
                        measures=['RMSE', 'MAE'],
                        cv=5,
                        return_train_measures=True,
                        verbose=True);
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std     
RMSE (testset)    1.2483  1.8867  1.1630  1.3902  1.3029  1.3982  0.2552  
MAE (testset)     0.9481  1.5240  0.8777  1.0560  0.9964  1.0804  0.2293  
RMSE (trainset)   1.2322  1.8780  1.1483  1.3662  1.2842  1.3818  0.2580  
MAE (trainset)    0.9311  1.5170  0.8614  1.0337  0.9770  1.0640  0.2334  
Fit time          4.25    4.34    4.31    4.66    4.41    4.39    0.14    
Test time         0.39    0.39    0.39    0.41    1.03    0.52    0.26    
```

In [283…
```
NMF_model_100_fit_time = np.sum(result['fit_time'])/len(result['fit_time'])
NMF_model_100_test_time = np.sum(result['test_time'])/len(result['test_time']
```

In [284…
```
NMF_model_100_rmse = np.sum(result['test_rmse'])/len(result['test_rmse'])
NMF_model_100_mae = np.sum(result['test_mae'])/len(result['test_mae'])
```

Part 6.5: Does overall accuracy change?

In [285…
```
sample_size = ['25%', '50%', '75%', '100%']
```

In [286…
```
item_model_mae = [item_model_25_mae, item_model_50_mae,
                  item_model_75_mae, item_model_100_mae]

item_model_rmse = [item_model_25_rmse, item_model_50_rmse,
                   item_model_75_rmse, item_model_100_rmse]
```

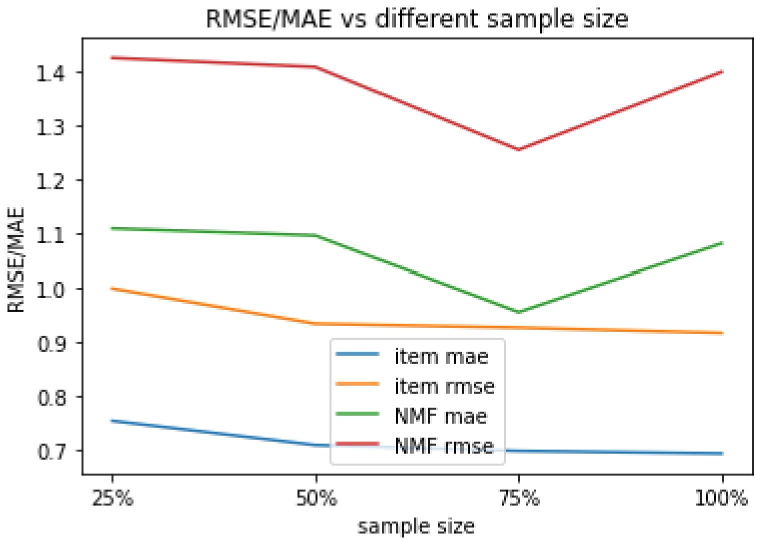
In [287…
```
NMF_model_mae = [NMF_model_25_mae, NMF_model_50_mae,
```

```
                 NMF_model_75_mae, NMF_model_100_mae]

NMF_model_rmse = [NMF_model_25_rmse, NMF_model_50_rmse,
                  NMF_model_75_rmse, NMF_model_100_rmse]
```

In [288…

```
fig, ax = plt.subplots()
ax.plot(sample_size, item_model_mae, label='item mae');
ax.plot(sample_size, item_model_rmse, label='item rmse');
ax.plot(sample_size, NMF_model_mae, label='NMF mae');
ax.plot(sample_size, NMF_model_rmse, label='NMF rmse');
ax.set_title('RMSE/MAE vs different sample size')
ax.set_xlabel('sample size')
ax.set_ylabel('RMSE/MAE')
ax.legend();
```



- According to the results above, the overall test accuracy of both models change as the sampling size increases.
- For item-based neighborhood model, both test RMSE and MAE decrease(monitonically) as the samping size increases.
- For NMF model, both test RMSE and MAE reach their minimum at 75% sampling size but slightly increase as the sampling size approaches 100%.

Part 6.6: What about the distribution of accuracy over users or items?

Explanation:

- Among the tested sample sizes and error methodologies, there is a relatively uniform distribution of model performance in terms of accuracy, with a momentary convergence at the 75% level. Item based mean absolute error performs the best across each of the tested models, and nonnegative matrix factorization with mean squared error performs the worst across the board. The split between them is relatively constant, again with a short kink improving the error for the NMF models at the 75% range.

Part 6.7: How does run-time scale with data size?

In [289…

```
sample_size = ['25%', '50%', '75%', '100%']
```

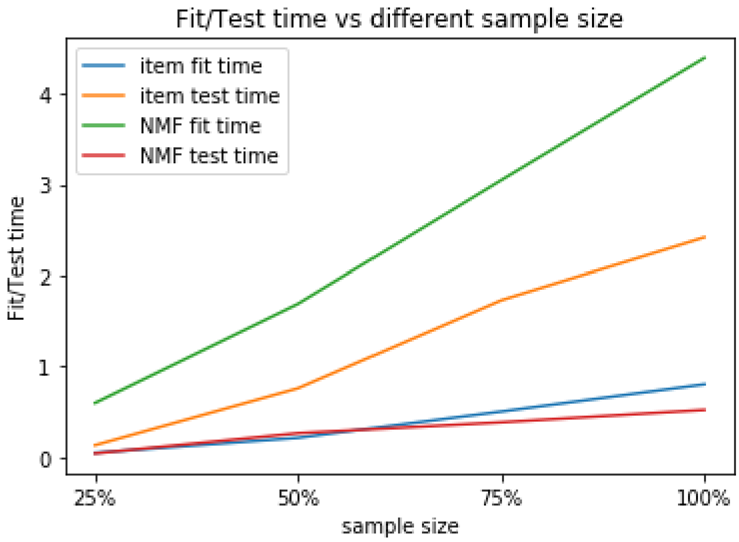
In [290…
```
item_model_fit_time = [item_model_25_fit_time,
                       item_model_50_fit_time,
                       item_model_75_fit_time,
                       item_model_100_fit_time]

item_model_test_time = [item_model_25_test_time,
                        item_model_50_test_time,
                        item_model_75_test_time,
                        item_model_100_test_time]
```

In [291…
```
NMF_model_fit_time = [NMF_model_25_fit_time,
                      NMF_model_50_fit_time,
                      NMF_model_75_fit_time,
                      NMF_model_100_fit_time]

NMF_model_test_time = [NMF_model_25_test_time,
                       NMF_model_50_test_time,
                       NMF_model_75_test_time,
                       NMF_model_100_test_time]
```

In [292…
```
fig, ax = plt.subplots()
ax.plot(sample_size, item_model_fit_time, label='item fit time');
ax.plot(sample_size, item_model_test_time, label='item test time');
ax.plot(sample_size, NMF_model_fit_time, label='NMF fit time');
ax.plot(sample_size, NMF_model_test_time, label='NMF test time');
ax.set_title('Fit/Test time vs different sample size')
ax.set_xlabel('sample size')
ax.set_ylabel('Fit/Test time')
ax.legend();
```



- According to the results above, both fit time and test time of NMF and item-based models increase as the sampling size increases.
- Especially for NMF, the increase of fit time of NMF is almost proportional to the increase of sampling size.

Part 7: How does your recommendation system meet your hypothetical objectives? Would you feel comfortable putting these solutions into production at a real company? What would be the potential watch outs?

Takeaways:

- Given this narrow-scoped dataset (just users, movies, and ratings), despite its large scale, we are proud of the hypothetical results we saw from our models. The sense-checks we performed in terms of seeing what movies are being recommended for a given user bear out that we are making sensible recommendations of popular movies, which is what we were looking to do from the very start.
- This workbook answers affirmatively the first business related questions addressed above about whether this model is viable and suits our company's objectives. We know it is effective, we know it makes good recommendations, and we know it has some issues - all production code does.
- We are happy to put the model into production with a few technical caveats (that we add measures for fairness in representation, add post-model logic for flagging movies that have already been watched or are not appropriate for a given user based on age or the like, etc).
- As it pertains to our business rules & aims, our current version achieves most of its objectives. We discount the unpopular films (another area for future development), we ensure novelty *within* a recommendation (i.e. won't recommend 'Lion King' twice) but do not ensure novelty at user level by flagging if a user has already seen and rated a film, we also do not have a method of indicating a trending movie and to weight it more heavily in the recommendation - another area for future dev, same story with recommending a partially watched movie - we might want to heavily weight it to remind a user to go back and complete their viewing but we have no way of flagging that with available data.

Part 8: Final remarks

Pros and Cons:

- The pros:
  1. It makes broadly appealing and sensible predictions/recommendations.
  2. It trains very quickly even on a laptop, so provided a corporate technical infrastructure, will both scale and be amply prepared for further fine-tuning of hyperparameters in that setting.
  3. We were able to collect a good deal of metadata about the models and the performance of training on data in this format, wherein we can be confident that we have chosen the strongest model for this dataset, and that our framework and pipeline is flexible for the addition of new features and other ways to boost performance (ensembling etc).
- The cons:
  1. There is some repetitive code in the demo here for processing different test/train sets and sampling data, but that repetition wouldn't be required in a production environment - this pipeline will transfer with minimal friction to a purposed data architecture.
  2. There is concern about the convergence of the models on the training data, we see much higher coverage on unseen user data for the model, because the model is less certain about those users. This means that we are either not including a representative sample of users in the train set for the model, or that the model is simply overfitting to the representative sample. We were willing to stomach a degree of overfitting because the predictions still got made in the same rank just

with different frequencies - this ought to self correct as the model is re-trained with more and more user data over time. If not , regularization techniques, ensembling with additional models, or adding new features are all ways we can mitigate this effect if need be.

In [ ]: