Collaborative Filtering

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [2]:
```
movies = pd.read_csv('data/movies_cleaned.csv')
movies.drop('Unnamed: 0', axis=1, inplace=True)

ratings = pd.read_csv('data/ratings_cleaned.csv')
ratings.drop('Unnamed: 0', axis=1, inplace=True)

tags = pd.read_csv('data/tags_cleaned.csv')
tags.drop('Unnamed: 0', axis=1, inplace=True)

ratings.head()
```

Out[2]:

	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 [3]:
```
ratings.shape
```

Out[3]:
```
(100836, 4)
```

In [4]:
```
ratings.drop('timestamp', axis=1, inplace=True)

from surprise import Reader, Dataset

reader = Reader()
data = Dataset.load_from_df(ratings, reader)
```

In [16]:
```
type(data)
```

Out[16]:
```
surprise.dataset.DatasetAutoFolds
```

Baseline models

In [5]:
```
from surprise.prediction_algorithms import BaselineOnly, KNNBasic, KNNWithMeans,
from surprise import accuracy
from surprise.model_selection import KFold
```

In [55]:
```
algos = [BaselineOnly(), KNNBasic(), KNNWithMeans(), SVD(), SlopeOne(), CoCluste
algo_names = ['BaselineOnly', 'KNNBasic', 'KNNWithMeans', 'SVD', 'SlopeOne', 'Co
kf = KFold(n_splits=5, random_state=123)
```

```python
for idx, algo in enumerate(algos):
    print('Algo:', algo_names[idx])
    print('\n')
    for idx, (trainset, testset) in enumerate(kf.split(data)):
        algo.fit(trainset)
        test_preds = algo.test(testset)

        # convert trainset into "testset" in order to get predictions
        train_preds = algo.test(trainset.build_testset())

        print('Fold {}'.format(idx+1))
        print('Train RMSE:', accuracy.rmse(train_preds, verbose=False))
        print('Test RMSE:', accuracy.rmse(test_preds, verbose=False))
    print('-----------------------------------------\n')
```

```
Algo: BaselineOnly


Estimating biases using als...
Fold 1
Train RMSE: 0.8359314343339654
Test RMSE: 0.8769476665685945
Estimating biases using als...
Fold 2
Train RMSE: 0.8347694743279065
Test RMSE: 0.8797867387844425
Estimating biases using als...
Fold 3
Train RMSE: 0.8372073513571463
Test RMSE: 0.8709396755927259
Estimating biases using als...
Fold 4
Train RMSE: 0.8393339217887553
Test RMSE: 0.8614796859985167
Estimating biases using als...
Fold 5
Train RMSE: 0.8373028893555643
Test RMSE: 0.8725240739858989
-----------------------------------------

Algo: KNNBasic


Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 1
Train RMSE: 0.7101389490773293
Test RMSE: 0.9518272007308126
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 2
Train RMSE: 0.7107380073610873
Test RMSE: 0.9516472785476531
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 3
Train RMSE: 0.7125696800833738
Test RMSE: 0.9422611786525384
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 4
Train RMSE: 0.7137725359959312
Test RMSE: 0.9294662355462576
```

```
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 5
Train RMSE: 0.7132733375244745
Test RMSE: 0.9450195604634429
-----------------------------------------

Algo: KNNWithMeans


Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 1
Train RMSE: 0.6856998705758436
Test RMSE: 0.8996614921912022
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 2
Train RMSE: 0.6858591546920433
Test RMSE: 0.9056668508748901
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 3
Train RMSE: 0.6871735120580375
Test RMSE: 0.8960753854308932
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 4
Train RMSE: 0.6888832232896717
Test RMSE: 0.8812066424179392
Computing the msd similarity matrix...
Done computing similarity matrix.
Fold 5
Train RMSE: 0.6887354619447559
Test RMSE: 0.8936271253291835
-----------------------------------------

Algo: SVD


Fold 1
Train RMSE: 0.6353158035087878
Test RMSE: 0.8757061853693824
Fold 2
Train RMSE: 0.6349792413283097
Test RMSE: 0.8821866057470671
Fold 3
Train RMSE: 0.6325193045283665
Test RMSE: 0.8699567982848373
Fold 4
Train RMSE: 0.6372383027262906
Test RMSE: 0.8621763805695373
Fold 5
Train RMSE: 0.6357104371158968
Test RMSE: 0.872886145562307
-----------------------------------------

Algo: SlopeOne


<ipython-input-55-e13bc1bfa5df>:9: DeprecationWarning: `np.int` is a deprecated
alias for the builtin `int`. To silence this warning, use `int` by itself. Doing
this will not modify any behavior and is safe. When replacing `np.int`, you may
wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish
to review your current use, check the release note link for additional informati
```

```
on.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdo
cs/release/1.20.0-notes.html#deprecations
  algo.fit(trainset)
Fold 1
Train RMSE: 0.5939534090379017
Test RMSE: 0.9064712969350329
Fold 2
Train RMSE: 0.5923854725648732
Test RMSE: 0.9072451288051778
Fold 3
Train RMSE: 0.5933649152829883
Test RMSE: 0.8982809483473887
Fold 4
Train RMSE: 0.5941299523001486
Test RMSE: 0.8872537132401754
Fold 5
Train RMSE: 0.5947064785946861
Test RMSE: 0.8969497079768711
-----------------------------------------

Algo: CoClustering


Fold 1
Train RMSE: 0.8149730924788463
Test RMSE: 0.9436859799970447
Fold 2
Train RMSE: 0.8142422949950948
Test RMSE: 0.951169728656384
Fold 3
Train RMSE: 0.8161835721564057
Test RMSE: 0.9436909327601742
Fold 4
Train RMSE: 0.8207156065748351
Test RMSE: 0.9292039748582993
Fold 5
Train RMSE: 0.8192091647443331
Test RMSE: 0.9355919104560565
-----------------------------------------
```

Of these prediction algorithms, SVD and BaselineOnly perform the best in terms of test set RMSE. SVD appaears to be overfit to the training data. I will continue to optimize both SVD and BaselineOnly.

Tuning & model selection

In [7]:
```
from surprise.model_selection import GridSearchCV
```

SVD

In [8]:
```
params = {'n_factors': [20, 50, 100, 150],
          'reg_all': [0.02, 0.05, 0.1]}

gs_svd = GridSearchCV(SVD, param_grid=params, n_jobs=-1)
gs_svd.fit(data)
```

In [9]:
```
print(gs_svd.best_params)
print(gs_svd.best_score)
```

```
{'rmse': {'n_factors': 50, 'reg_all': 0.05}, 'mae': {'n_factors': 150, 'reg_al
l': 0.05}}
{'rmse': 0.8685569017111574, 'mae': 0.6677608987007162}
```

In [30]:
```
params = {'n_factors': [30, 40, 50, 60, 70],
          'reg_all': [0.03, 0.05, 0.07]}

gs_svd2 = GridSearchCV(SVD, param_grid=params, n_jobs=-1)
gs_svd2.fit(data)
```

In [31]:
```
print(gs_svd2.best_params)
print(gs_svd2.best_score)
```

```
{'rmse': {'n_factors': 50, 'reg_all': 0.05}, 'mae': {'n_factors': 40, 'reg_all':
0.03}}
{'rmse': 0.8688949378180041, 'mae': 0.6676795148278568}
```

The second SVD gridsearch chooses the same values for `n_factors` and `reg_all` as the first grid search without improvement in RMSE.

BaselineOnly (ALS)

In [48]:
```
params = {'bsl_options': {'method': ['als'],
                          'n_epochs': [5, 10, 15],
                          'reg_u': [10, 15, 20],
                          'reg_i': [5, 10, 15]}}

gs_bsl_only = GridSearchCV(BaselineOnly, param_grid=params, n_jobs=-1)
gs_bsl_only.fit(data)
```

In [49]:
```
print(gs_bsl_only.best_params)
print(gs_bsl_only.best_score)
```

```
{'rmse': {'bsl_options': {'method': 'als', 'n_epochs': 15, 'reg_u': 10, 'reg_i':
5}}, 'mae': {'bsl_options': {'method': 'als', 'n_epochs': 15, 'reg_u': 10, 'reg_
i': 5}}}
{'rmse': 0.8663971733011129, 'mae': 0.6663383473579636}
```

In [50]:
```
params = {'bsl_options': {'method': ['als'],
                          'n_epochs': [20, 50, 100],
                          'reg_u': [2, 4, 8, 10, 12],
                          'reg_i': [1, 3, 5, 7]}}

gs_bsl_only2 = GridSearchCV(BaselineOnly, param_grid=params, n_jobs=-1)
gs_bsl_only2.fit(data)
```

In [51]:
```
print(gs_bsl_only2.best_params)
print(gs_bsl_only2.best_score)
```

```
{'rmse': {'bsl_options': {'method': 'als', 'n_epochs': 100, 'reg_u': 4, 'reg_i':
3}}, 'mae': {'bsl_options': {'method': 'als', 'n_epochs': 100, 'reg_u': 2, 'reg_
i': 3}}}
{'rmse': 0.863728619261542, 'mae': 0.6629538626379057}
```

In [56]:
```
params = {'bsl_options': {'method': ['als'],
                          'n_epochs': [20, 50],
                          'reg_u': [2, 4, 8, 10],
                          'reg_i': [1, 3, 5, 7]}}

gs_bsl_only3 = GridSearchCV(BaselineOnly, param_grid=params, n_jobs=-1)
gs_bsl_only3.fit(data)
```

In [57]:
```
print(gs_bsl_only3.best_params)
print(gs_bsl_only3.best_score)
```

```
{'rmse': {'bsl_options': {'method': 'als', 'n_epochs': 50, 'reg_u': 4, 'reg_i':
3}}, 'mae': {'bsl_options': {'method': 'als', 'n_epochs': 50, 'reg_u': 2, 'reg_
i': 3}}}
{'rmse': 0.8647400877308197, 'mae': 0.6637614096730935}
```

It seems that GridSearch will choose the higher number of epochs given the option, but with little improvement in RMSE. There is no reason to choose 100 epochs over 50 with such an insignificant gain in performance. Though we could use a model with even fewer than 50 epochs, I will use the best model from the third BaselineOnly GridSearch to make rating predictions. This RMSE is lower than that of the most tuned SVD model, but not by much. Furthermore, BaselineOnly showed no signs of overfitting initially.