

# knn

December 13, 2020

[ ]:

## 1 Item-based Collaborative Filtering

Collaborative filtering considers users' opinion on different products and recommends the best products based on the products' previous rankings and the opinion of other similar types of users.

Here we focus on non-probabilistic collaborative filtering algorithms, which can be divided to two categories: **memory-based** and **model-based**.

Memory-based algorithm is essentially linear algebra calculation and can be implemented with **k-nearest neighbors**. Model-based algorithm involves matrix factorization, and will be detailed in the next notebook.

**Item-based** collaborative filtering is the common practice in recommender systems. The intuition is to generate predictions based on similarities between items.

Item-based collaborative filtering was developed by Amazon. It is faster when there are more users than items. It is also more stable because the average rating received by an item usually doesn't change as quickly as the average rating given by a user to different items.

The advantage of item-based collaborative filtering is that it does not require knowledge about the product.

[ ]:

### 1.0.1 import requirements

```
[168]: # import library
import pandas as pd
import numpy as np
import scipy as sp

import matplotlib.pyplot as plt
import seaborn as sns

from collections import defaultdict
from collections import Counter
```

Source: [surprise documentation](#)

```
[169]: from surprise import Dataset
from surprise import Reader
from surprise import KNNWithMeans
from surprise.model_selection import GridSearchCV
from surprise import accuracy
```

```
[ ]:
```

```
[171]: # load data
df = pd.read_csv('ratings_item0.csv', index_col=0)
df.head(3)
```

```
[171]:      uid      bid  rating
10  276746  0425115801      0
11  276746  0449006522      0
12  276746  0553561618      0
```

```
[172]: df.shape
```

```
[172]: (456182, 3)
```

```
[173]: num_users = len(set(df['uid']))
num_books = len(set(df['bid']))
print(f'There are {num_users} users and {num_books} books in this dataset.')
```

There are 13808 users and 18318 books in this dataset.

```
[ ]:
```

## 1.0.2 data loading

```
[22]: reader = Reader(rating_scale=(1,10))
```

```
[23]: data = Dataset.load_from_df(df[['uid', 'bid', 'rating']], reader)
```

```
[ ]:
```

```
[178]: # param_grid = {
#     'bsl_options': {'method': ['als', 'sgd'],
#                     #'reg_i': [10],
#                     #'reg_u': [15],
#                     #'reg': [0.02],
#                     #'learning_rate': [0.005],
#                     #'n_epochs': [10, 20]},
#     'k': [3, 5, 10, 20],
#     'sim_options': {'name': ['cosine', 'msd', 'pearson', 'pearson_baseline'],
#                     'min_support': [1, 3],
```

```
# 'user_based': [True, False]}
# }
```

```
[179]: # gs = GridSearchCV(KNNWithMeans,
#                       param_grid,
#                       measures=['rmse', 'mae'],
#                       cv=3)
```

```
[180]: #gs.fit(data)
```

[illegible]









[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]



[illegible]



[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```

Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.

```

```

[184]: # # best RMSE score
      # print(gs.best_score)

```

```
{'rmse': 1.6651286278605182, 'mae': 1.2377969578142525}
```

```

[203]: best_score = {'rmse': 1.6651286278605182, 'mae': 1.2377969578142525}
      print(best_score)

```

```
{'rmse': 1.6651286278605182, 'mae': 1.2377969578142525}
```

```

[182]: # # combination of parameters that gave the best RMSE score
      # print(gs.best_params['rmse'])

```

```
{'bsl_options': {'method': 'als', 'n_epochs': 10}, 'k': 20, 'sim_options':
{'name': 'cosine', 'min_support': 1, 'user_based': False}}
```

```

[201]: best_params = {'bsl_options': {'method': 'als', 'n_epochs': 10}, 'k': 20,
      ↪ 'sim_options': {'name': 'cosine', 'min_support': 1, 'user_based': False}}
      print(best_params)

```

```
{'bsl_options': {'method': 'als', 'n_epochs': 10}, 'k': 20, 'sim_options':
{'name': 'cosine', 'min_support': 1, 'user_based': False}}
```

```

[183]: # results_df = pd.DataFrame.from_dict(gs.cv_results)
      # results_df.head()

```

```

[183]: split0_test_rmse  split1_test_rmse  split2_test_rmse  mean_test_rmse  \
0          1.767441          1.782065          1.775103          1.774870
1          1.732224          1.735080          1.739080          1.735462
2          1.807058          1.821138          1.828133          1.818776
3          1.723783          1.739371          1.732963          1.732039
4          1.763404          1.794554          1.775311          1.777756

      std_test_rmse  rank_test_rmse  split0_test_mae  split1_test_mae  \
0          0.005973          186          1.327541          1.339127
1          0.002812          129          1.291303          1.295970

```

2	0.008764	251	1.351508	1.345143
3	0.006397	116	1.293963	1.298640
4	0.012834	190	1.318364	1.340988

	split2_test_mae	mean_test_mae	std_test_mae	rank_test_mae	mean_fit_time \
0	1.325365	1.330678	0.006040	217	0.801336
1	1.295773	1.294349	0.002155	96	0.294262
2	1.357642	1.351431	0.005103	251	0.406097
3	1.300495	1.297699	0.002748	133	0.098307
4	1.320052	1.326468	0.010290	207	0.308598

	std_fit_time	mean_test_time	std_test_time \
0	0.015515	0.417113	0.011231
1	0.018206	0.306368	0.053522
2	0.085957	0.406642	0.038261
3	0.008631	0.270384	0.015879
4	0.009363	0.463069	0.052258

	params \
0	{'bsl_options': {'method': 'als', 'n_epochs': ...
1	{'bsl_options': {'method': 'als', 'n_epochs': ...
2	{'bsl_options': {'method': 'als', 'n_epochs': ...
3	{'bsl_options': {'method': 'als', 'n_epochs': ...
4	{'bsl_options': {'method': 'als', 'n_epochs': ...

	param_bsl_options	param_k \
0	{'method': 'als', 'n_epochs': 10}	3
1	{'method': 'als', 'n_epochs': 10}	3
2	{'method': 'als', 'n_epochs': 10}	3
3	{'method': 'als', 'n_epochs': 10}	3
4	{'method': 'als', 'n_epochs': 10}	3

	param_sim_options
0	{'name': 'cosine', 'min_support': 1, 'user_bas...
1	{'name': 'cosine', 'min_support': 1, 'user_bas...
2	{'name': 'cosine', 'min_support': 3, 'user_bas...
3	{'name': 'cosine', 'min_support': 3, 'user_bas...
4	{'name': 'msd', 'min_support': 1, 'user_based'...

```
[185]: #results_df.to_csv('knn_results.csv')
```

Save the hypertuning results so next time I don't have to run that again and can directly use the results.

```
[ ]:
```



### 1.0.3 evaluation

#### 1.0.4 MAE (mean absolute error)

Mean Average Error (MAE) does not give any bias to extrema in error terms. If there are outliers or large error terms, it will weigh those equally to the other predictions. Therefore, MAE should be preferred when looking toward rating accuracy when you're not really looking toward the importance of outliers. To get a holistic view or representation of the Recommender System, use MAE.

#### 1.0.5 RMSE (root mean squared error)

Root Mean Squared Error tends to disproportionately penalize large errors as the residual (error term) is squared. This means RMSE is more prone to being affected by outliers or bad predictions.

MAE can only accurately recreate 0.8 of the data set. RSME is a lot more mathematically convenient whenever calculating distance, gradient, or other metrics. That's why most cost functions in Machine Learning avoid using MAE and rather use sum of squared errors or Root Means Squared Error.

By definition, RMSE will never be as small as MAE. [This paper](#) establishes that both metrics are useful in evaluating algorithms.

```
[233]: #results_df = pd.read_csv('knn_results.csv')
```

```
[225]: # results_df = results_df[
#      ['split0_test_rmse', 'rank_test_rmse', 'params',
#      'param_bsl_options', 'param_k', 'param_sim_options']]
```

```
[226]: # results_df = pd.concat([
#      results_df.drop(['param_bsl_options'], axis=1),
#      results_df['param_bsl_options'].apply(pd.Series)
#      ], axis=1)
```

```
[228]: # results_df = pd.concat([
#      results_df.drop(['param_sim_options'], axis=1),
#      results_df['param_sim_options'].apply(pd.Series)
#      ], axis=1)
```

```
[231]: #results_df.to_csv('knn_results_cleaned.csv')
```

```
[235]: results_df = pd.read_csv('knn_results_cleaned.csv', index_col=0)
```

```
[258]: results_df.sort_values(by='rank_test_rmse')[:5]
```

```
[258]:      split0_test_rmse  rank_test_rmse  \
177          1.663016          1
241          1.663016          2
49           1.663016          3
113          1.663016          4
```

161

1.668153

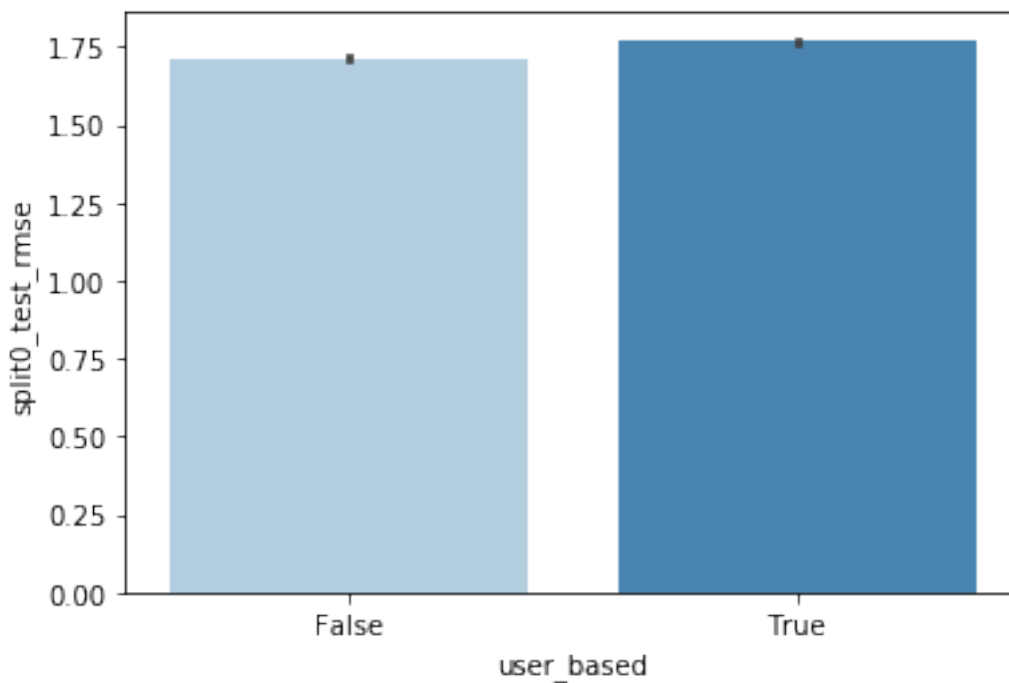
5

		params	param_k	method	\
177	{'bsl_options': {'method': 'sgd', 'n_epochs': ...		20	sgd	
241	{'bsl_options': {'method': 'sgd', 'n_epochs': ...		20	sgd	
49	{'bsl_options': {'method': 'als', 'n_epochs': ...		20	als	
113	{'bsl_options': {'method': 'als', 'n_epochs': ...		20	als	
161	{'bsl_options': {'method': 'sgd', 'n_epochs': ...		10	sgd	

	n_epochs	name	min_support	user_based
177	10	cosine	1	False
241	20	cosine	1	False
49	10	cosine	1	False
113	20	cosine	1	False
161	10	cosine	1	False

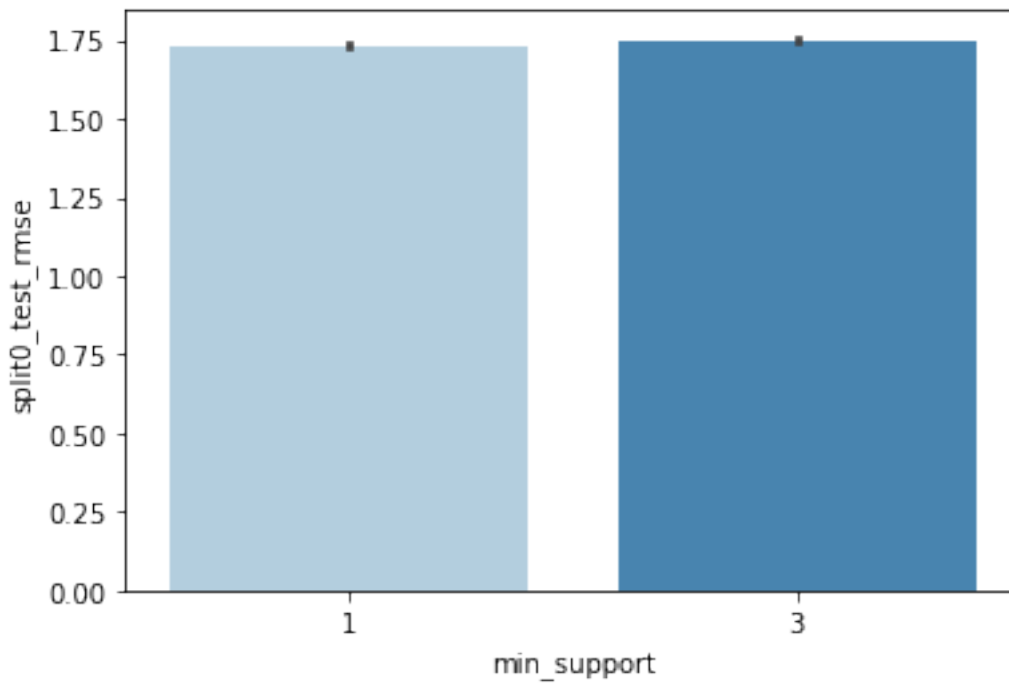
```
[328]: sns.barplot(x=results_df['user_based'], y=results_df['split0_test_rmse'],
    ↪palette='Blues')
```

```
[328]: <matplotlib.axes._subplots.AxesSubplot at 0x13b7b0df0>
```



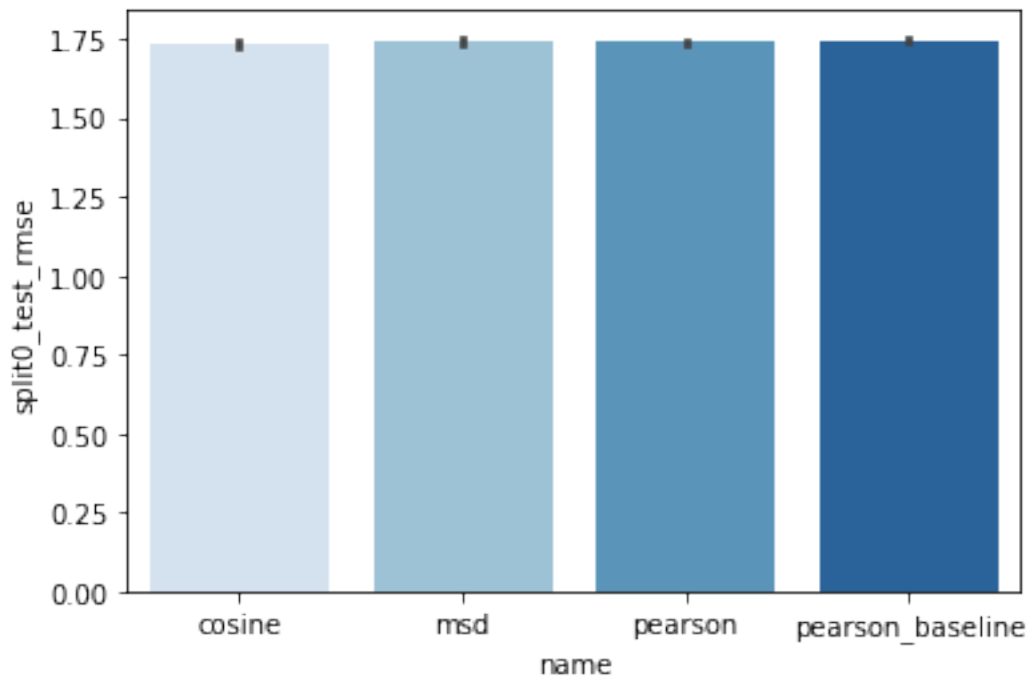
```
[329]: sns.barplot(x=results_df['min_support'], y=results_df['split0_test_rmse'],
    ↪palette='Blues')
```

[329]: <matplotlib.axes.\_subplots.AxesSubplot at 0x13b530fa0>



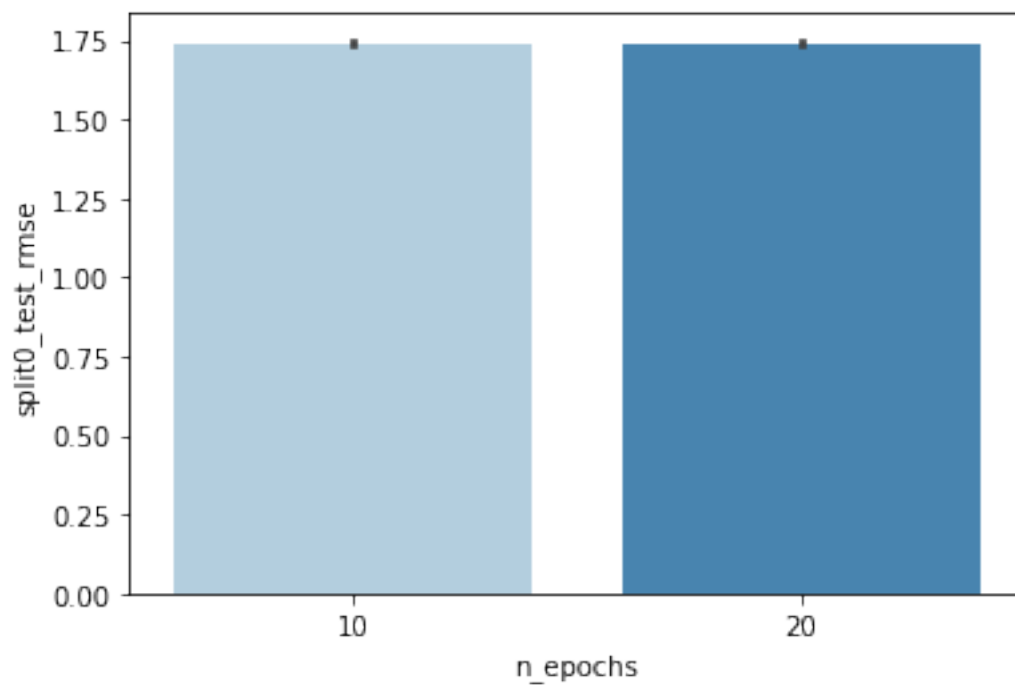
```
[330]: sns.barplot(x=results_df['name'], y=results_df['split0_test_rmse'],  
                  ↪palette='Blues')
```

[330]: <matplotlib.axes.\_subplots.AxesSubplot at 0x13b398eb0>



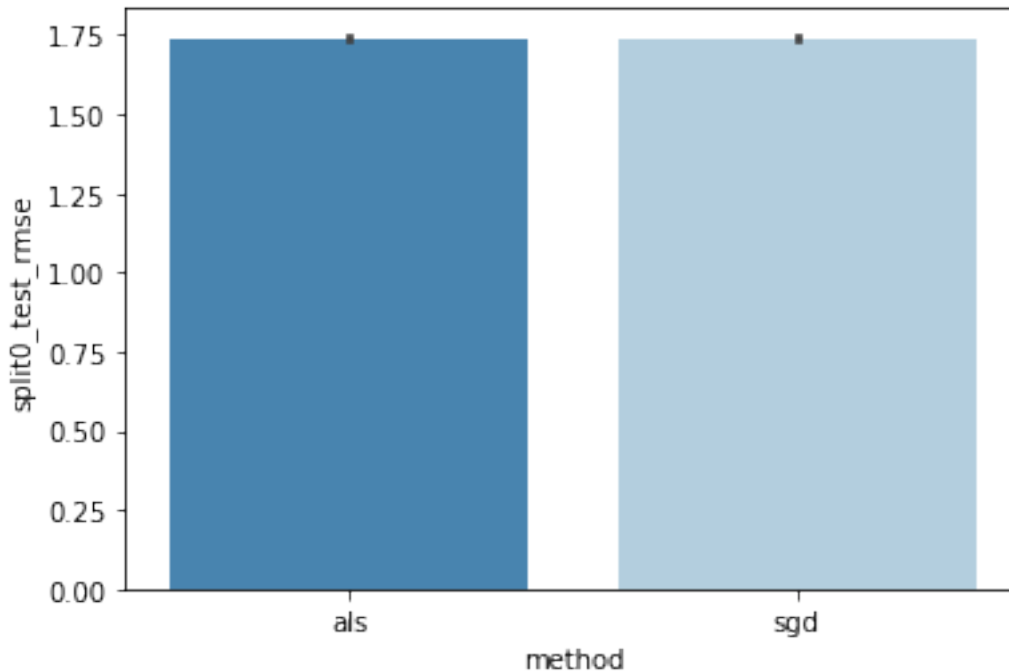
```
[333]: sns.barplot(x=results_df['n_epochs'], y=results_df['split0_test_rmse'],  
               palette='Blues')
```

```
[333]: <matplotlib.axes._subplots.AxesSubplot at 0x13a6add00>
```



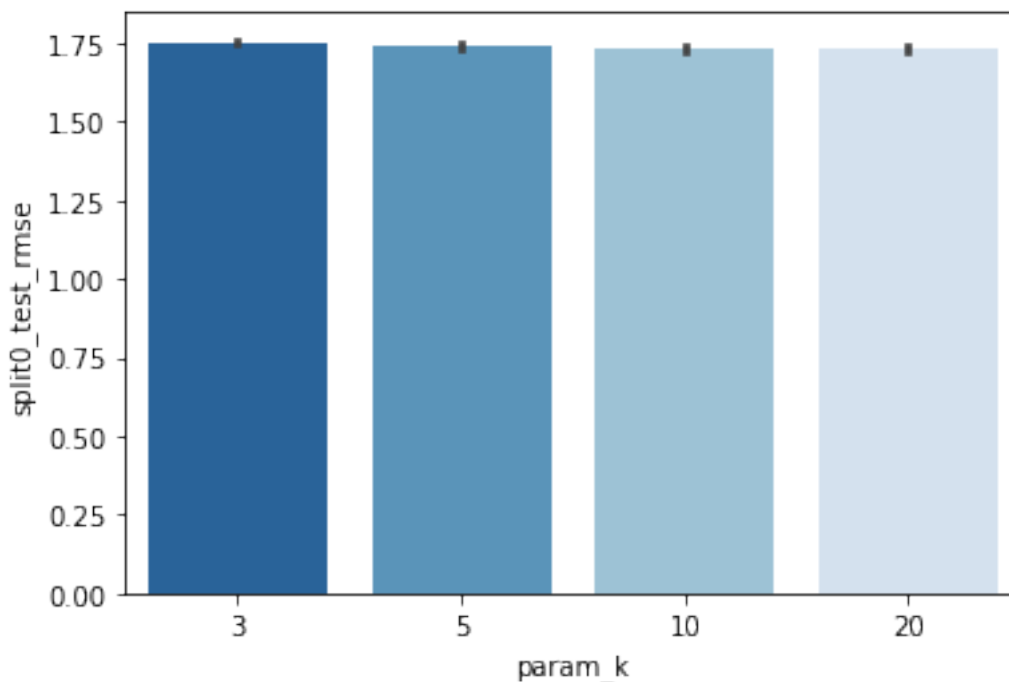
```
[332]: sns.barplot(x=results_df['method'], y=results_df['split0_test_rmse'],  
                  ↪palette='Blues_r')
```

```
[332]: <matplotlib.axes._subplots.AxesSubplot at 0x13b47f6d0>
```



```
[331]: sns.barplot(x=results_df['param_k'], y=results_df['split0_test_rmse'],  
                  ↪palette='Blues_r')
```

```
[331]: <matplotlib.axes._subplots.AxesSubplot at 0x13a5d02e0>
```



```
[254]: results_df.sort_values(by='rank_test_rmse')
```

```
[254]:
```

	split0_test_rmse	rank_test_rmse	\
177	1.663016	1	
241	1.663016	2	
49	1.663016	3	
113	1.663016	4	
161	1.668153	5	
..	...	...	
66	1.807058	252	
198	1.812107	253	
70	1.812107	254	
6	1.812107	255	
134	1.812107	256	

	params	param_k	method	\
177	{'bsl_options': {'method': 'sgd', 'n_epochs': ...	20	sgd	
241	{'bsl_options': {'method': 'sgd', 'n_epochs': ...	20	sgd	
49	{'bsl_options': {'method': 'als', 'n_epochs': ...	20	als	
113	{'bsl_options': {'method': 'als', 'n_epochs': ...	20	als	
161	{'bsl_options': {'method': 'sgd', 'n_epochs': ...	10	sgd	
..	...	...	...	
66	{'bsl_options': {'method': 'als', 'n_epochs': ...	3	als	
198	{'bsl_options': {'method': 'sgd', 'n_epochs': ...	3	sgd	

```

70  {'bsl_options': {'method': 'als', 'n_epochs': ...      3    als
6   {'bsl_options': {'method': 'als', 'n_epochs': ...      3    als
134 {'bsl_options': {'method': 'sgd', 'n_epochs': ...      3    sgd

```

	n_epochs	name	min_support	user_based
177	10	cosine	1	False
241	20	cosine	1	False
49	10	cosine	1	False
113	20	cosine	1	False
161	10	cosine	1	False
..	...	...	...	...
66	20	cosine	3	True
198	20	msd	3	True
70	20	msd	3	True
6	10	msd	3	True
134	10	msd	3	True

[256 rows x 9 columns]

[ ]:

[ ]:

[ ]:

## 1.1 get top-N recommendations for each user

Source: [documentation](#)

We first train an SVD algorithm on the whole dataset, and then predict all the ratings for the pairs (user, item) that are not in the training set. We then retrieve the top-10 prediction for each user.

```

[188]: def get_top_n(predictions, n=10):
        """
        Return the top-N recommendation for each user from a set of predictions.

        Args:
            predictions(list of Prediction objects): The list of predictions, as
                returned by the test method of an algorithm.
            n(int): The number of recommendation to output for each user. Default_
                is 10.

        Returns:
            A dict where keys are user (row) ids and values are lists of tuples:
                [(raw item id, rating estimation), ...] of size n.
        """

```

```

# First map the predictions to each user.
top_n = defaultdict(list)
for uid, iid, true_r, est, _ in predictions:
    top_n[uid].append((iid, est))

# Then sort the predictions for each user and retrieve the k highest ones.
for uid, user_ratings in top_n.items():
    user_ratings.sort(key=lambda x: x[1], reverse=True)
    top_n[uid] = user_ratings[:n]

return top_n

```

[ ]:

[213]: trainset, testset = train\_test\_split(data, test\_size=.25)

[214]: algo = KNNWithMeans(k=20,  
bsl\_options={'method':'als', 'n\_epochs':10},  
sim\_options={'name':'cosine', 'min\_support':1,  
'user\_based':False})

[215]: predictions = algo.fit(trainset).test(testset)

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

[ ]:

[189]: #top\_10 = get\_top\_n(predictions, n=10)

[326]: # # Print the recommended items for each user  
# for uid, user\_ratings in top\_10.items():  
# pass  
# #print(uid, [bid for (bid, \_) in user\_ratings])

[206]: test\_df = pd.DataFrame(testset, columns=['uid', 'bid', 'rating'])  
test\_df.head(3)

[206]:

	uid	bid	rating
0	212849	0684853515	9.0
1	115003	0375700757	8.0
2	11676	0425163407	8.0

[281]: #test\_df[test\_df['uid']==115003]

[282]: #top\_10[115003]



```
[283]: # bought = set(test_df[test_df['uid']==115003]['bid'])
# predicted = set([bid for bid, rating in top_10[115003]])
# bought.intersection(predicted)
```

```
[286]: # # the rate our recommended books in user's bought list
# len(bought.intersection(predicted)) / len(predicted)
```

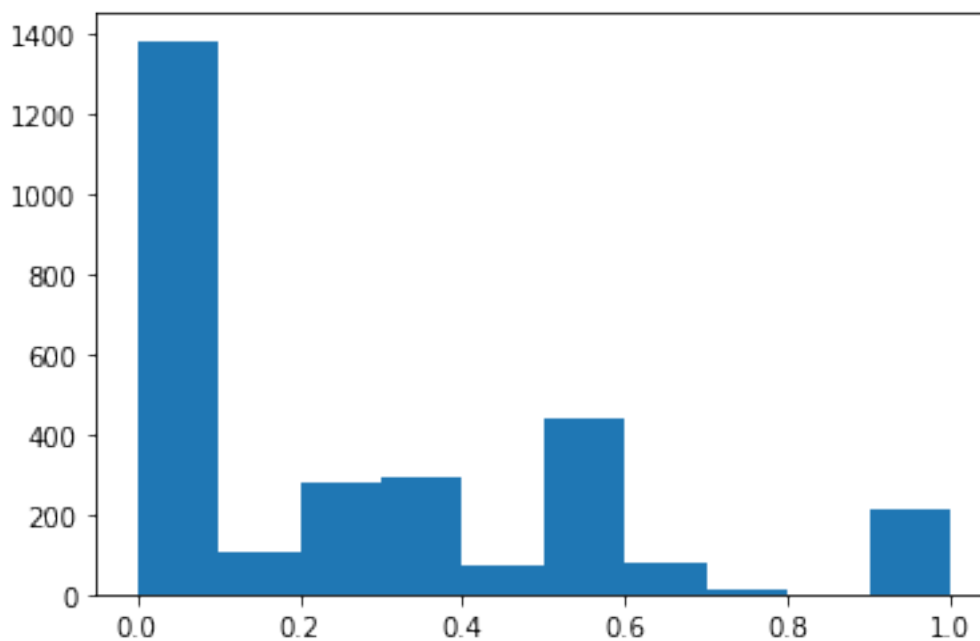
```
[ ]:
```

```
[327]: top_10 = get_top_n(predictions, n=10)
accuracy = []

# Print the recommended items for each user
for uid, user_ratings in top_10.items():
    bought = set(test_df[test_df['uid']==uid]['bid'])
    predicted = set([bid for bid, rating in user_ratings])
    rate = len(bought.intersection(predicted)) / len(predicted)
    accuracy.append(rate)
print(sum(accuracy)/len(accuracy))
```

0.24570465736843541

```
[325]: plt.hist(accuracy)
plt.show()
```



```
[ ]:
```

This means that based on our current recommender system, when we predict them their top 10 books, we can accurately recommend around 25% of the books they would end up actually buying.

```
[ ]:
```

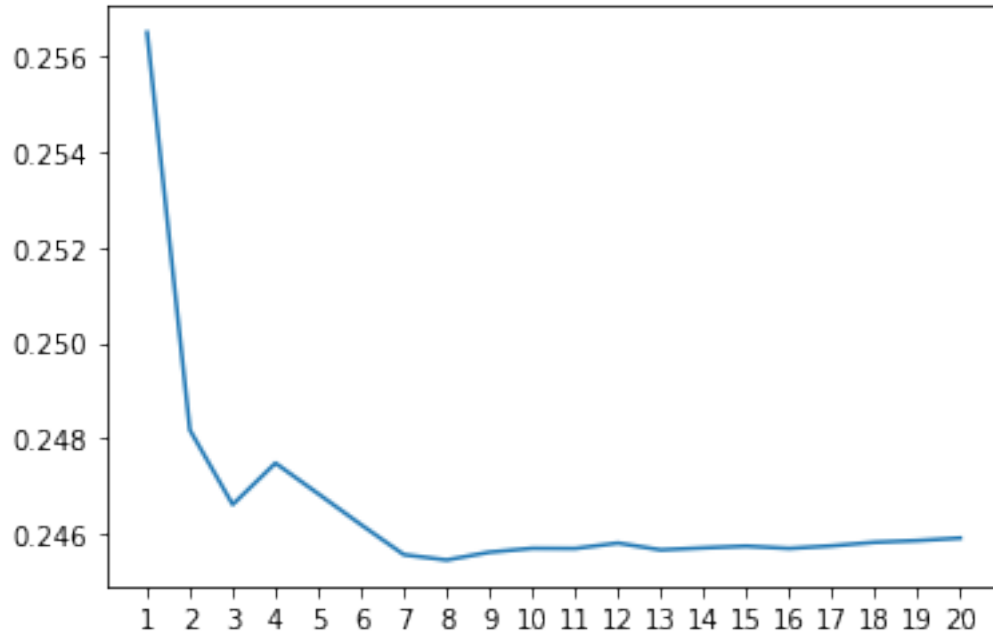
```
[298]: n = np.arange(1,21,1)
avg_accuracy = []

for i in n:
    top_n = get_top_n(predictions, n=i)
    accuracy = []

    for uid, user_ratings in top_n.items():
        bought = set(test_df[test_df['uid']==uid]['bid'])
        predicted = set([bid for bid, rating in user_ratings])
        rate = len(bought.intersection(predicted)) / len(predicted)
        accuracy.append(rate)

    avg_accuracy.append(sum(accuracy)/len(accuracy))

[317]: plt.plot(avg_accuracy)
plt.xticks(np.arange(0,20,1),n)
plt.show()
```



```
[ ]:
```

Interestingly, our recommendation accuracy does not yield better results when we predict more items, and there is an optimal result accuracy when we recommend 4 items to users.

[ ]:

## 1.2 Issues in collaborative filtering

- Sparseness in user-item matrix ( $\text{sparsity} = 1 - |R|/|I| * |U|$ )
- Cold start for users and items

Next, we would try to resolve these issues using matrix factorization.

[ ]: