cf-02-rzepinskip-grid_search-analysis

January 21, 2019

1 Important

`make data/processed/ratings-train.csv` has to be run before running any notebook cell

2 Imports

```
In [1]: import pandas as pd
```

3 Grid search

Search for optimal parameters values was conducted using GridSearch method. The whole process base only on training dataset, to avoid introducing a bias into test procedure.

Helper methods to prepare data and compare results:

```
In [2]: def flatten_dicts(df):
            if 'param_sim_options' in df.columns:
                df_sim = df['param_sim_options'].apply(lambda x :
        dict(eval(x))).apply(pd.Series)
                df = pd.concat([df, df_sim], axis=1).drop('param_sim_options', axis=1)
            if 'param_bsl_options' in df.columns:
                df_sim = df['param_bsl_options'].apply(lambda x :
        dict(eval(x))).apply(pd.Series)
                df = pd.concat([df, df_sim], axis=1).drop('param_bsl_options', axis=1)
            return df

        def select_cols(df, cols):
            df = flatten_dicts(df)
            cols = ['mean_test_rmse'] + cols + ['mean_fit_time', 'mean_test_time']
            return df[cols]

        def compare(df, col):
            return df.groupby(col)[['mean_test_rmse', 'mean_fit_time', 'mean_test_time']
                                ].median().sort_values('mean_test_rmse')
```

3.1 KNN

3.1.1 Choosing similarity metric

Available metrics are: `- Cosine: - Mean Squared Difference: - Pearson: - Pearson with baseline:`

Because number of users is much higher than number of items, we use item-based similarity.

Shrinkage parameter can be specified for Pearson Baseline to avoid overfitting when only few ratings are available. In our dataset there are always at least 8 ratings for books, so there is no need to tune this parameter.

```
In [3]: df_sim_metric = pd.read_csv("../results/knn-parameters-search-sim_metric.csv",
        index_col='rank_test_rmse')
        df_sim_metric = select_cols(df_sim_metric, ['name'])
        compare(df_sim_metric, 'name')
```

Out[3]:

name	mean_test_rmse	mean_fit_time	mean_test_time
pearson_baseline	0.801832	60.299604	186.840445
pearson	0.839508	99.129660	204.765986
msd	0.842515	35.385960	212.359089
cosine	0.845685	72.453772	193.802783

Pearson baseline methods achieves the best results due to the fact, that it take into account the baselines. Further explained in section 2.1 of "Factor in the Neighbors: Scalable and Accurate Collaborative Filtering" by Koren.

3.1.2 Choosing baselines estimates method

Available methods are: - `SGD`: Stochastic Gradient Descent - `ALS`: Alternating Least Squares

```
In [4]: df_baselines = pd.read_csv("../results/knn-parameters-search-baselines.csv",
        index_col='rank_test_rmse')
        df_baselines = select_cols(df_baselines, ['method', 'n_epochs', 'reg', 'learning_rate',
        'reg_i', 'reg_u'])
        compare(df_baselines, 'method')
```

Out[4]:

method	mean_test_rmse	mean_fit_time	mean_test_time
als	0.801832	64.515593	153.113881
sgd	0.801860	79.026545	180.148679

ALS and SGD achieve comparable results with deafult parameters, but ALS is trained faster. Therefore, we choose ALS for tuning.

3.1.3 Choosing neighbors count

Available parameters are: - `k`: maximal number of neighbors to take into account; default value 40 - `min_support`: minimal number of similar users between neighbor and current item for calculating similarity isntead of returning 0; default value 1

```
In [5]: df_neighbors = pd.read_csv("../results/knn-parameters-search-neighbors.csv",
        index_col='rank_test_rmse')
        df_neighbors = select_cols(df_neighbors, ['param_k', 'min_support'])
        df_neighbors.head()
```

Out[5]:

rank_test_rmse	mean_test_rmse	param_k	min_support	mean_fit_time	\
1	0.801160	20	1	65.863054	
2	0.801507	20	3	65.328067	

```
3                    0.801832       40             1       69.702970
4                    0.802074       60             1       66.119544
5                    0.802269       40             3       65.148404

                 mean_test_time
rank_test_rmse
1                    154.485534
2                    150.975937
3                    205.608619
4                    195.468733
5                    194.164147
```

```
In [6]: compare(df_neighbors, ['param_k'])
```

```
Out[6]:          mean_test_rmse  mean_fit_time  mean_test_time
param_k
20             0.801507      65.328067      150.975937
40             0.802269      65.148404      194.164147
60             0.802501      64.909069      184.853892
```

Taking into consideration smaller number of neighbors seems to benefit the model's accuracy.

```
In [7]: compare(df_neighbors, ['min_support'])
```

```
Out[7]:              mean_test_rmse  mean_fit_time  mean_test_time
min_support
1                  0.801832      66.119544      195.468733
3                  0.802269      65.148404      184.853892
5                  0.803479      59.592568      166.467101
```

As expected, any value for score is better than 0.

3.1.4 Choosing regularization parameters

```
In [8]: df_knn_reg = pd.read_csv("../results/knn-parameters-search-reg.csv",
        index_col='rank_test_rmse')
        df_knn_reg = select_cols(df_knn_reg, ['n_epochs', 'reg_i', 'reg_u'])
        df_knn_reg.head()
```

```
Out[8]:                 mean_test_rmse  n_epochs  reg_i  reg_u  mean_fit_time  \
rank_test_rmse
1                    0.801084        10      5     10      67.401383
2                    0.801085        15      5     10      71.608517
3                    0.801102        10     10     10      69.800121
4                    0.801102        15     10     10      74.266051
5                    0.801173        15     20     10      67.920909

                 mean_test_time
rank_test_rmse
1                    160.205643
2                    154.410492
```

```
3                   171.306709
4                   189.423243
5                   165.916787
```

```
In [9]: compare(df_knn_reg, ['n_epochs', 'reg_i', 'reg_u'])
```

Out[9]:

n_epochs	reg_i	reg_u	mean_test_rmse	mean_fit_time	mean_test_time
10	5	10	0.801084	67.401383	160.205643
15	5	10	0.801085	71.608517	154.410492
10	10	10	0.801102	69.800121	171.306709
15	10	10	0.801102	74.266051	189.423243
	20	10	0.801173	67.920909	165.916787
10	20	10	0.801176	70.721758	158.806097
	5	15	0.801372	68.040582	153.536440
15	5	15	0.801374	75.412613	159.427819
10	10	15	0.801389	60.129907	150.864869
15	10	15	0.801390	63.606051	173.669571
10	20	15	0.801455	69.331677	157.575999
15	20	15	0.801455	66.874159	163.984486
10	5	20	0.801667	68.326596	152.332470
15	5	20	0.801670	66.239547	166.337182
10	10	20	0.801684	72.032163	187.006729
15	10	20	0.801685	59.888889	164.533296
10	20	20	0.801747	66.807248	144.758057
15	20	20	0.801748	72.352360	146.918359

There are no huge differences between obtained results, so we stick to the defaults.

3.1.5 Final settings

```
In [10]: knn_params = {'bsl_options': {'method': ['als'],
                                       'reg_i': [10],
                                       'reg_u': [15],
                                       'n_epochs': [10]},
                       'k': [30],
                       'sim_options': {'name': ['pearson_baseline'],
                                       'min_support': [1],
                                       'user_based': [False],
                                       'shrinkage': [100]},
                       'verbose': [False]}
```

3.2 SVD

We opted for SVD instead of SVD++. The latter requires far more time for training phase(5 vs 150 minutes) and scores better by only 0.01 points(0.82 vs 0.81).

3.2.1 Choosing factors number

```
In [11]: df_factors = pd.read_csv("../results/svd-parameters-search-factors.csv",
                                  index_col='rank_test_rmse')
         df_factors = select_cols(df_factors, ['param_n_factors'])
         compare(df_factors, 'param_n_factors')
```

Out[11]:

param_n_factors	mean_test_rmse	mean_fit_time	mean_test_time
50	0.835244	139.449435	29.343737
100	0.836891	224.073945	26.769742
200	0.839633	388.132762	27.209588
500	0.846550	1246.017054	28.506237

Surprisingly, higher number of factors does not result in better accuracy.

3.2.2 Choosing regularization parameters

```
In [12]: df_init = pd.read_csv("../results/svd-parameters-search-init.csv",
         index_col='rank_test_rmse')
         df_init = select_cols(df_init, ['param_init_mean', 'param_init_std_dev'])
         df_init.head()
```

Out[12]:

rank_test_rmse	mean_test_rmse	param_init_mean	param_init_std_dev	mean_fit_time	mean_test_time
1	0.823991	0.1	0.05	206.355935	9.811689
2	0.824589	0.0	0.05	229.683039	11.310694
3	0.833249	0.3	0.05	209.813560	10.087036
4	0.836442	0.1	0.10	232.116656	10.440973
5	0.836891	0.0	0.10	238.212949	12.279941

As the ratings mean is much greater than average point of the scale(3.9 vs 2.5) we can assume than starting from median closer to real one would yield better results. That turned out to be true.

```
In [13]: df_svd_reg = pd.read_csv("../results/svd-parameters-search-reg.csv",
         index_col='rank_test_rmse')
         df_svd_reg = select_cols(df_svd_reg, ['param_n_epochs', 'param_lr_all',
         'param_reg_all'])
         df_svd_reg.head()
```

Out[13]:

rank_test_rmse	mean_test_rmse	param_n_epochs	param_lr_all	param_reg_all	mean_fit_time	mean_test_time
1	0.820582	25	0.005	0.02	238.362410	9.419897
2	0.823991	20	0.005	0.02		
3	0.831536	20	0.010	0.02		
4	0.836851	25	0.050	0.10		
5	0.837979	20	0.050	0.10		

2	191.720921	9.544194
3	190.925779	9.479615
4	239.002583	9.534884
5	190.633800	9.459400

Default parameters for surprise library were already adjusted for 1-5 rating scale. Therefore different parameter values give worse results.

3.2.3 Final settings

```
In [14]: svd_params = {'n_factors': [100],
                       'biased': [True],
                       'init_mean': [0.1],
                       'init_std_dev': [0.05],
                       'n_epochs': [25],
                       'lr_all': [0.005],
                       'reg_all': [0.02],
                       'random_state': [44]}
```