Big Data Final Project: Recommender Systems

Eileen Cho, Tin Luu, Ademola Oladosu
`{ec3636, tbl245, ao1584}@nyu.edu`

1 Basic Recommendation System

1.1 Implementation

We create our recommendation model using the ALS method in Spark to learn latent factor representations for users and items. To ensure that we have a good model without over-fitting, we train our model on the cf_train.parquet data set using different values for the hyper-parameters regParam, alpha, and rank, and evaluate the top 500 item recommendations for each user against the cf_validation.parquet data set.

1.2 Hyper-parameter Tuning

The script to train and tune hyper-parameters is found in EC_training.py. The final set of values selected for tuning are shown below. By spanning orders of magnitude we can get a reasonable initial hypothesis as to what parameters are appropriate for our model.

- regParam: [0.01, 0.1, 1, 10]
- alpha: [0.1, 1, 10, 100]
- rank: [100]

These were selected based on the outputs from previously failed iterations of implementing our system. We evaluate the model performance by using PySpark methods precisionAt and meanAveragePrecision (weighted by recommendation order).The mean average precision is the average of the precision at every position in the list of predictions for the users. This is a useful metric because both we attempt to predict the occurrence and position of an item to be recommended. Although we also calculate the precision, which trends quite closely with the mean average precision, it is the MAP that we use as our critical evaluation metric.

Although we explored ranks higher than 100, we have excluded these results because of the marginal improvements that were observed. The improved results did not justify the increased computation time. We also saw from previous results during implementation that models with rank below 100 regardless of value of other hyper-parameters produced lower performing models. Therefore we fix rank at 100. In total, we considered sixteen possibilities of hyper-parameters.

1.3 Evaluation

After these trials were run, the best set of hyper-parameters corresponded to regParam = 10, alpha = 100, and rank = 100. The full set of results is shown in Table 1. We use the script EC$_test.pytoevaluateourfinalmodelagainstthedataset.Whenweevaluatedagainstthetestdata, wegetaprecisionof 0.016, andamean

The next steps would be to refine the three parameters to be more precise in presenting the optimal set of values. The optimal regParam and Alpha lie at the edge of the hyper-parameters we elected to tune so it is entirely possible that the true optimum involves values beyond the scope of our tuning. It is in these areas that we would focus further analysis.

1.4 Concessions and Trade-offs

One of the primary constraints during model development and analysis was the size of the training data set. One strategy we utilized for cutting down the size of the training data set was to select only rows containing users who were present in the validation and test file. This could impact the model's performance because users who were trained on only have partial histories.

During model evaluation, there were two techniques we used during our model analysis. The preferred technique was to train the model, save it, then load the model from the test file. However, due to challenges and instability of the dumbo cluster, we have also created a version to train the model, and directly evaluate against the test data.

Table 1: Hyper-parameter Tuning Results on Validation Data

regParam	Alpha	Rank	Precision	Mean Average Precision
0.01	0.1	100	0.006235	0.029588
0.01	1	100	0.007169	0.033590
0.01	10	100	0.009415	0.047175
0.01	100	100	0.010881	0.050562
0.1	0.1	100	0.007266	0.028886
0.1	1	100	0.007217	0.034701
0.1	10	100	0.009428	0.047760
0.1	100	100	0.010922	0.051425
1	0.1	100	0.003812	0.003992
1	1	100	0.007167	0.037777
1	10	100	0.009537	0.051585
1	100	100	0.011151	0.055370
10	0.1	100	0.000202	0.000124
10	1	100	0.000141	0.000415
10	10	100	0.008217	0.038276
10	**100**	**100**	**0.011552**	**0.058491**

Table 2: Evaluation of Best Model On Test Data

regParam	Alpha	Rank	Precision	Mean Average Precision
10	100	100	0.0116	0.0584

2 Extension 1: Alternative Model Formulations

The goal was to determine if performing certain transformations to the "count" column would result in a significant impact on performance evaluation. We evaluated three transformations. The first compressed the count column by taking the natural logarithm of the recorded value. The second compressed the data set by dropping all rows where the count was one. Given we have no count values of zero in our data set, we show this as dropping counts less than or equal to 1. The third transformation dropped all rows where the count was less than or equal to 2. With each transformation, a model was obtained using the best set of hyper-parameters from the hyper-parameter tuning that was described in the previous section. These results can then be compared to the pre-transformed result to evaluate if there was any improvement or deterioration observed from the transformation. The results are shown in the table below.

Table 3: Evaluation of Best Model On Test Data

regParam	Alpha	Rank	Precision	Mean Average Precision
10	100	100	0.0116	0.0584

Table 4: Evaluation of Count Transformation on Test Data

Type of Transformation	Precision	Mean Average Precision
None	0.0116	0.0584
Log	0.0077	0.0394
Drop Count ≤ 1	0.0085	0.0397
Drop Count ≤ 2	0.0072	0.0317

As expected, the best performance in terms of precision and MAP is the pre-transformed data set. There is quite a drop-off in precision and mean average precision when the log of the count is considered and when the lowest counts are dropped. However, among these transformations, the difference in performance is not as large in our case. This suggests that if we are satisfied with the lower result for mean average precision, we can likely save time on computation by restricting the rows that we consider in our model development.

3 Extension 2: Fast Search

When producing a model and evaluating the results, the limiting step is the recommendation for all users. The `annoy` library creates an index for approximate nearest neighbors search. Ultimately, we expect that the tree structure employed by the annoy library will help reduce the time required to generate recommendations for all users. Using our best set of hyper-parameters, we wish to evaluate the time difference in generating both models.

We attempted to generate recommendations for all 100,000 users with `annoy` but the high computational processing cost for this task on our local machine did not allow this task to complete. Henceforth, we sampled 1000 users from the test set and generate recommendations for each. By restricting our prediction task to a smaller size, we had the ability to explore a wider range of hyper-parameter settings in the `annoy` library.

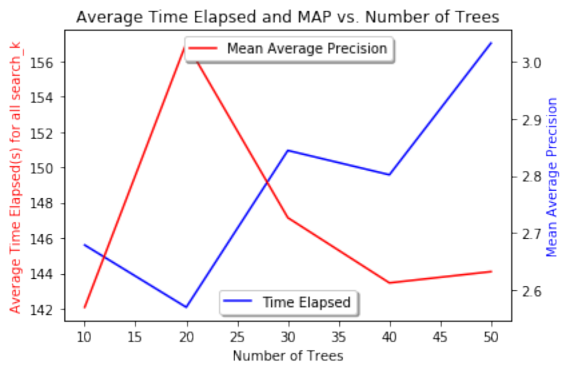
One of the manipulated variables available in the annoy library is the number of trees built in our forest, which is given by n_trees. More trees gives higher precision when querying. The second variable that can be tuned is $search\_k$, which can be increased to provide more accurate results. The trade-off is that doing this will increase the run time. The maximum run time is at a $search\_k$ value of -1. The results of the permutations are shown below.

- n_trees: [10, 20, 30, 40, 50]
- search_k: [-1, 10, 50, 100]

Table 5: Hyper-parameter Tuning of n_trees and search_k from Annoy Library (1000 users)

n_trees	search_k	Time Elapsed (s)	Precision at 500	Mean Average Precision (X10 ˆ -5)
N/A	N/A	232.67	0.0118	6540
10	-1	145.60	.000092	2.57030
10	10	154.00	.000092	2.57030
10	50	160.99	.000092	2.57030
20	100	145.64	.000114	3.03188
20	10	138.14	0.000114	3.03188
20	50	146.70	0.000114	3.03188
20	**100**	**137.84**	**0.000114**	**3.03188**
30	-1	145.39	.000102	2.72674
30	10	145.81	0.000102	2.72674
30	50	172.74	0.000102	2.72674
30	100	139.88	0.000102	2.72674
40	-1	155.96	.000102	2.61286
40	10	146.26	0.000102	2.61286
40	50	146.49	0.000102	2.61286
40	100	149.60	0.000102	2.61286
50	-1	161.50	.000104	2.63251
50	10	153.14	0.000104	2.63251
50	50	160.53	0.000104	2.63251
50	100	152.96	0.000104	2.63251

Among our results, the lowest time observed was at n_trees of 20 and search_k of 100. This elapsed time of 138 seconds is a 40% reduction from the baseline result that is shown in the first row. In our set of hyper-parameters, within the range of parameters selected, the largest impact was seen by increasing the number of trees. To 5 decimal places, we did not observe meaningful differences in our time elapsed when search_k was varied. Below is a graph of a summary of our results. For our data set, n_trees appears to be optimal in terms of maximizing MAP and minimizing the time elapsed.



Given that we only time the results for 1000 users, we may find the impact to be more meaningful on a substantially larger data set or when search_k is increased far beyond 100.

Appendix

Contributions

- All members contributed to writing the report
- Eileen - Implementing Basic Recommender System, Implementing Extension 2
- Tin - Implementing the initial stages of building the basic recommender system, implemented the baseline models
- Ademola - Developing hyper-parameter tuning structure, Implementing Extension 1, Literature Review for Annoy/NMSLIB Libraries, Python Graphics Generation

Installation and Running Instructions

- Basic Recommender System - We have two working versions, one where we train and tune our model in EC_training.py, and load the model in EC_test.py to evaluate against test data, and another version that in EC_train_and_test.py which trains and tunes a model, and directly continues on to evaluate the recommendations against the test data.
  - **Training and Tuning Model** - `spark-submit EC_training.py root_path_to_data training_data validation_data test_data path_to_save_model boolean_for_tuning`
  - This program reads in training, validation, and test datasets in order to determine which user rows are absolutely necessary for creating the model. For the sake of testing the entire pipeline without going through hyper-parameter tuning, there is a parameter ‘tuning‘ which defaults to False. Setting it to True allows for hyper-parameter tuning.
  - **Testing Model** - `spark-submit EC_test.py path_to_test_data path_to_indexers path_to_load_model`
  - This program loads a saved model file, a saved indexer (string indexers), and evaluates recommendations against the test data set.
  - **Training and Testing**- `spark-submit EC_train_and_test.py path_to_train_data path_to_validation_data path_to_test_data path_to_model_file boolean_for_tuning`
  - This program is a complete pipeline of the previous two programs, for use when the dumbo cluster has problems with a loaded model file.
- Extension 1: Alternative Model Formulations
  - **Testing different count transformations** - `spark-submit EC_training_alt_models.py root_path_to_all_data train_data validation_data test_data model_file1 model_file2 model_file3 model_file4 boolean_for_tuning`
  - This program takes in paths to all data sets, and trains a model on data after count transformations, using the hyper-parameters from our best model from the basic recommender system
- Extension 2: Fast Search
  - This extension requires use of the external library `annoy`.
  - **Testing Query Time and Precision** - `spark-submit EC_ext2.py path_to_test_file path_to_indexers path_to_model, limit`
  - This program takes the path to test data, path to indexers (string indexers) and path to model in order to extract the necessary information of our trained brute force model. The parameter limit determines how many users we wish to query for.