DSGA1004-Big Data Final Project

Anshan HE (ah4734), Shuting GU (sg5686)

Center for Data Science, New York University
May 2019

Abstract

In the final project, we built and evaluated a recommender system using the tools we have learned. The final collaborative filer model gave a satisfying performance with the *MAP* = 0.08226 and the *NDCG at 500* = 0.28141. We also implemented an extension to modify the count data based on the built model. The main strategies that we applied are dropping low count values. It is possible that we can achieve the similar or even better accuracy with relatively higher efficiency.

1 Introduction

In this final project, we tried to build a collaborative filter to personally recommend tracks to different users.
Latent factor models are popular for collaborative filtering. Using matrix factorization, we can construct user-factor and item-factor matrices to fill in the missing entries of a user-item association matrix. And the alternative least squares (ALS) model is commonly used to learn the latent factor and it solves the problem in parallel.
In train, validation and test sets, the count number for each user-track pair is provided. As an implicit feedback, the count number is a weak signal. Fortunately, `spark.ml` provides an approach to deal with such data. And we included it when training the ALS model.
The large amount of data in the training set may create low efficiency, so we will firstly discuss the downsampling method. Then we transferred the users and tracks identifiers from strings to indices, which are proper input for the model. There are three parameters, `rank`, `regParam`, and `alpha`, that we selected to tune. The detailed methods and the final results will be discussed in the section 4. With the optimal group of parameters, we trained an ALS model and used the test set to test its performance. In the last section, we implemented an extension, *alternative model formulations*, by mainly modifying counts. For the strategies that we applied, we evaluated their performance and impact.

2 Downsampling

After loading the parquet file containing training data, we created a DataFrame including a total of 49824519 records and 1129318 distinct users. In order to save time and memory, we applied a downsampling method by using `down_sample.py`.
We selected all the distinct users from `cf_train.parquet` and kept the last 110K users unchanged, since they only have partial histories and those records are important for later validation and testing processes. For other users whose histories are full, we randomly chose partial distinct users. Finally, we joined the two subsets, collected all their records from `cf_train.parquet`, and obtained a smaller train sample with 14530236 records and 329318 distinct users.

3 Feature Transformation

We noticed that the user and track identifiers in the DataFrames created by loading the parquet files are strings. In order to turn raw DataFrames into ML-friendly features, `StringIndxer` is applied to transform `user_id` and `track_id` columns of string labels to columns of label indices to fit the Spark's ALS model.
Two `StringIndexer` transformations are used to transfer the columns `user_id` and `track_id` separately. These two transformations are then bundled into a pipeline. We fitted

the pipeline to the union of training, validation and test DataFrames to ensure that all included user and track identifiers can be transformed to a proper type and the same user and the same track will share the same index after the transformation.
We saved the transformed DataFrames as parquet files and used them directly and repeatedly in the following parameter tuning, model training and model testing processes to save time. The transformation is realized by using `string_index.py`.

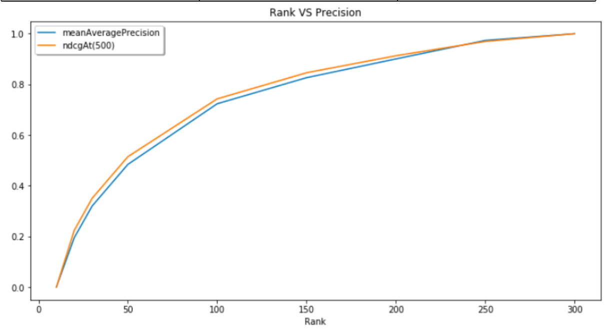
4 Hyper-Parameter Tuning

In this section, we are going to tune three parameters of the ALS model: `rank`, `regParam`, and `alpha`. We used the `parameter_tuning.py` to achieve our goal.
In the ALS model, `rank` is an integer represents the number of latent factors in the model (defaults to 10); `regParam` is a float specifies the regularization parameter in ALS (defaults to 1.0); `alpha` is a float and it is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations (defaults to 1.0).
Grid search is implemented to find the best group of parameters. We used the downsampled training set to train ALS models with different parameter combinations, and used the validation set to test the model performance. The metrics `meanAveragePrecision` and `ndcgAt(500)` from `RankingMetrics` were chosen to measure the performance of each model and select the best combination. `meanAveragePrecision` returns the mean average precision (*MAP*) of all the queries and `ndcgAt(500)` returns the average of normalized discounted cumulative gain at the first 500 ranking positions. Both metrics take the order of the recommendations into account so they are more meaningful and suitable for the model selection.
The selection process and the optimal value we selected for each parameter will be presented below. To show the changing trends of different metrics in one plot, we applied min-max normalization to normalize the *MAP* and *NDCG at 500*.

4.1 Hyper-Parameter Tuning - `rank`

Several values of `rank` were tested with `regParam` and `alpha` fixed to be around the optimal values. The results of two metrics are shown in the table and line plot below.

`rank`	MAP	ndcgAt(500)
10	0.03016707	0.14698863
20	0.03981180	0.17670947
30	0.04599848	0.19349303
50	0.05413389	0.21529963
100	0.06599067	0.24561153
150	0.07108274	0.25927341
200	0.07474571	0.26818761
250	0.07836448	0.27568612
300	0.07967843	0.27973984

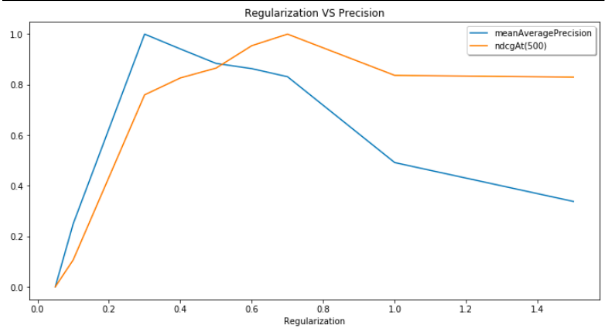


As what we observed from the plot, when `rank` increases, both the *MAP* and *NDCG at 500* increase. However, the growth slows down when `rank` is more than 150.
The value of `rank` represents the number of latent factors in the model. So it also decides the sizes of user-factor and item-factor matrices. The larger the value of `rank` is, the more elements in matrices need to be calculated, which means the process time will be longer. Moreover, since we only used a small part of data to train the model, the high rank may cause overfitting problem. Considering both accuracy and efficiency, we selected `rank` to be 250.

4.2 Hyper-Parameter Tuning - `regParam`

Several values of `regParam` were tried with `rank` and `alpha` fixed to be around their optimal values. The performance is shown in the table and the line plot below.

regParam	MAP	ndcgAt(500)
0.05	0.06543845	0.24433033
0.1	0.06559391	0.24448739
0.3	0.06606307	0.24545482
0.4	0.06602642	0.24555373
0.5	0.06599067	0.24561153
0.6	0.06597775	0.24574287
0.7	0.06595772	0.24581046
1.0	0.06574548	0.24556881
1.5	0.06564944	0.24555838

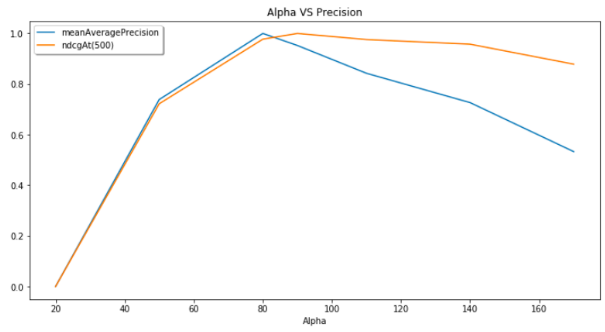


From the table and the plot, we noticed that with `rank` and `alpha` fixed, both the *MAP* and *NDCG at 500* are maximized when `regParam` is around 0.3 to 0.6. Therefore, 0.5 has been selected to be the final value we used for `regParam`.

4.3 Hyper-Parameter Tuning - ***`alpha`***

We tried several values of `alpha` with `rank` and `regParam` fixed. The results are shown in the table and line plot below.

alpha	MAP	ndcgAt(500)
20	0.06543845	0.23479697
50	0.06551243	0.24270195
80	0.06631434	0.24549563
90	0.06616828	0.24574404
110	0.06583092	0.24547995
140	0.06547494	0.24527925
170	0.06488000	0.24441451



The plot indicates that if `regParam` and `rank` are fixed, both the *MAP* and *NDCG at 500* are peaked when `alpha` is around 80 to 100. Thus, 90 has been selected to be the final value we used for `alpha`.

5 Training of Recommender System

Using the optimal group of parameters $(\texttt{rank}, \texttt{regParam}, \texttt{alpha}) = (250, 0.5, 90)$ we trained the ALS model with the whole training dataset.
As for the parameter `implicitPrefs`, we set it to be `True` so we can handle the implicit feedback (i.e. count) with the scaling parameter `alpha`. The resulted ALS model is saved for the following testing step.

6 Testing of Recommender System

We used the test set to evaluate the trained model. The evaluation is based on the predictions of the top 500 tracks for each user and `RankingMetrics` will be used as an evaluator of the model.
We used `recommendForUserSubset` to return the predicted top 500 tracks recommended for each user in the test set. Meanwhile, we produced the other DataFrame containing information of an actual rank of tracks for each user ordered by `count`.
Then we joined two DataFrames and changed it to a resilient distributed dataset (RDD), which can be used as the input of `RankingMetrics`. In order to evaluate the performance of our recommender system, we produced the values for three metrics: *Precision at 500*, *MAP* and *NDCG at 500*. The results are shown below:

$$\text{Precision at 500} = 0.01476482$$
$$\text{MAP} = 0.08226456$$
$$\text{NDCG at 500} = 0.28141166$$

Using the *Precision at 500*, we measures how many of the first 500 tracks recommended are in the set of actual tracks that users have frequent interactions with. But since it does not take the order of the recommendations into account, the *MAP* is a better measure for our system. And we noticed that when we consider the order of the recommendations, the metric value is higher. For *NDCG at 500*, it considers not only the order, but also an averaged result across all users instead of an individual. It returns a value 0.28, which is satisfying and demonstrates that our recommender system performed well.

7 Extension - *Alternative model formualtions*

Due to the implicit feedback of ALS model, we applied several modification strategies to the `count` data, hoping to improve the efficiency and accuracy of the collaborative filter.

7.1 Drop Record with Low Count

Records with low values of `count` might affect the precision of the recommender system. We left the last 110K users' records unchanged, drop all the records with `count = 1` from the remaining training set by using `modify_count.py`. A smaller training set means that we can save much more time when tuning parameter and training model without down-sampling. After a simple hyper-parameter tuning, the results we achieved are

$$\text{Precision at 500} = 0.01165981$$
$$\text{MAP} = 0.06109574$$
$$\text{NDCG at 500} = 0.22032057$$

It is possible that we can achieve the similar or even better results after implementing the grid search when tuning the hyper-parameters.

7.2 Drop Track of Low Total Count

In the previous strategy, we deleted the tracks with low count related to single users. However, these tracks might have interaction with many different users and have a high total counts. So instead of dropping low `count` records directly, we dropped tracks with low total counts using `modify_track.py` to make comparison. We chose to drop tracks with total counts less than 5000. However, the results are worse:

$$\text{Precision at 500} = 0.00693824$$
$$\text{MAP} = 0.03727064$$
$$\text{NDCG at 500} = 0.13624693$$

The worse results may caused by two reasons:
1). When we deleted the tracks with low total account, we deleted some records related to the last 110K users. For those users, low-count records are important for the prediction.
2). From the previous section, we knew `alpha` has a high value. This means that even if we only have one interaction between a user and track, the confidence is higher than that of the unknown data.

7.3 Count Transformation

Aiming at improving training accuracy, we considered to apply transformation (e.g., log compression and normalization) to the original training data set directly or the training set after previous modifications.
Ideally, after each modification, we will get a new training set with new data and need to go through the hyper-parameter tuning process again. However, due to the difficulties with the dumbo cluster and personal devices' limitation, we were unable to finish all the steps.

8 Conclusion and Result

In the report, we built an ALS model using `spark.ml`. After downsampling, feature transformations, and hyper-parameter tuning, we trained the baseline collaborative filter model. Using the test set, we evaluated the model and get satisfying results. In the extension, we applied different strategies to modify the count data. It is possible that after suitable modification we can get the similar or even better result by using a smaller training set with a part of data. This will increase the efficiency for model training.

9 Contribution

Both group members contributed equally to every part of the project.

References

[1] H. Yifan, K. Yehuda and V. Chris, *Collaborative Fil-tering for Implicit Feedback Datasets*, December 2001, 2008 Eighth IEEE International Conference on Data Mining.

[2] *Collaborative Filtering - RDD-based API*, 2014, Retrieved from https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html

[3] *Evaluation Metrics - RDD-based API*, 2014, Retrieved from https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html