Recommendation System on Million Song Dataset using Alternating Least Square Algorithm

ZIYI XIE, NYU Center for Data Science, US
LETIAN JIANG, NYU Center for Data Science, US

This paper explains how we use Alternating Least Square algorithm to the truncated version of Million Song Dataset as a recommendation system.

Additional Key Words and Phrases: ALS, Million Song Dataset, Apache Spark, big data

ACM Reference Format:
Ziyi Xie and Letian Jiang. 2018. Recommendation System on Million Song Dataset using Alternating Least Square Algorithm. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In this project, we are experimenting a recommendation system based on ALS(Alternating Least Square) algorithm over a user-item dataset using Apache Spark. The goal of such system is to recommend users a set of items that they might like. The dataset used here is from Million Song Dataset [1]. The dataset contains multiple features, but we are not using all of them for the basic implementation of recommendation system using ALS. The goal of the basic model for this project is to make track recommendations for users. In addition to the basic recommendation model using ALS, we build an extension that does fast search with Annoy package.

2 DATASET

The slice of MSD used for the basic model has a schema that contains user id, track id, and count, where count represents the number of times that a certain user listen to a certain track as an implicit feedback. The data we are using also consist of three parts: train, validation, and test. In table 1 is a short summary of the slice of dataset used in this project. Based on the number of entries of each partition, we have an approximate ratio of train: validation: test = 97%: 0.3%: 2.7%

3 MODEL AND EVALUATION

3.1 Model

The ALS algorithm is a model included in the machine learning package of Apache Spark. ALS is a collaborative filtering model that is popular for building recommendation systems. It is a matrix factorization algorithm that uses alternating least square loss and weighted lambda regularization [3] . A matrix factorization algorithm is where the user and items

Manuscript submitted to ACM

Table 1. data partition statistics

partition	number of entries	count of distinct users	count of distinct tracks	minimum count	maximum count
train	49,824,519	1,129,318	385,371	1	9,667
validation	138,938	10,000	49,994	1	787
test	1,368,430	100,000	158,956	1	2,265

to be recommended matrix R is factorized into a user-to-feature matrix U and a item-to-feature matrix V. Matrix R comes directly from the transformation of training data into matrix form, and U, V will be learned. Both U and V will be trained to produce a matrix that is as close to R as possible by learning the latent factors, which are the matrix elements in U and V. Diagram 1 shows the relationship between R, U, V. In U and V. For the ALS, we will also use the cold start strategy drop, where unseen users during testing will get dropped from prediction result. While this will cause some problems in the model deployment in real world, this is the best choice for the ALS model from Apache Spark, which doesn't offer other solutions for cold start problem. The ALS model will be trained using different latent factor ranks and regularization parameters, where the rank refers to the dimension of U, V and regularization suppresses over-fitting. Both parameter will also be fine tuned on evaluation data.

3.2 Evaluation

To evaluate the trained model, we consider a comparison between the top 500 tracks recommended to individual users and the actual tracks the individual users would listen in validation data or test data.

We are using two evaluation methods: Mean Average Precision(MAP) and Precision at 500. We have a list of users $U_i = u_1, u_2, ...$, and corresponding track list in order of count $T_i = t_1, t_2, ...$ The model takes U_i as input and produces $R_i = r_1, r_2, ...$ as recommendations in order of count as well. We want T and R to be in descending order of counts because the evaluation metrics weighs on the order.

For precision at 500, this metric calculates how many of the top 500 recommendations R are in T. For mean average precision, this metric calculates the number of recommendations R in T with a penalty of mismatch that is higher for higher counts and lower for lower counts.

3.3 Fine Tuning

Fine tuning will be implemented on the rank of latent factors and the size of regularization parameters. The choice of rank includes: 1 ,5 , 10, 15, 20, 25, 30, 35,40,45,50. The choice of regularization parameter, includes 0.001, 0.01, 0.05, 0.1, 1, 5, 10.

4 RESULTS

The model is first fitted on train set, and then evaluated on validation set. It turns out that larger than 1 regularization parameter would yield low evaluation scores, so we won't include those regularization parameter values in the plot. The results of tuning is shown in Fig 1.

For both evaluation metric MAP and Precision at 500, the figure indicates that the best combination of parameters is rank = 50 and regParam = 0.5 . So this set of parameters will be used to evaluate the test set.

An observation is that for the majority of regularization parameter, the increasing the number of rank would make the model to perform better in both metrics. However, if we look at the case where regParam is 0.001, we see possible over-fitting is happening when rank increases from 15 to 20. This should be caused by the small size of regularization

parameter. Therefore we might conclude that as we keep increase rank, over-fitting will happen for model with any regularization Parameter. So we want to use 0.5 regularization parameter with increasing rank of the model to see what would happen. We decide to train models with rank 100, 150, 200 and 250, but the cluster we run the training on is not producing an result before the deadline of this project.

Therefore, we settled with the ALS model with regularization parameter 0.5 and rank 50. The result of running this model, trained with full training data, on the test set yield MAP = 0.0352 and Precision at 500 = 0.00875.

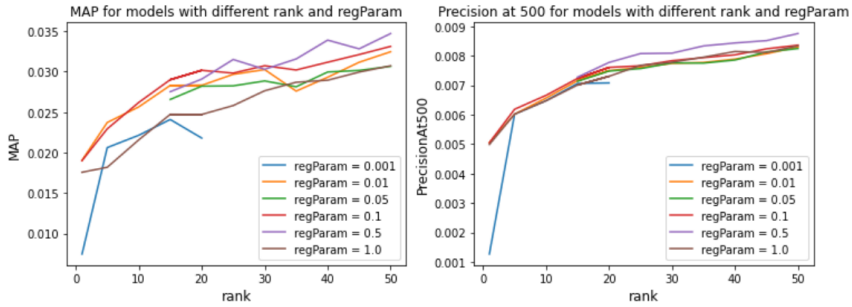

Fig. 1. Parameter tuning results. First trained on full train data, then evaluated on validation data. Results for two evaluation scores are shown.

5 EXTENFTER RUNNING MATRIX FACTORIZATION ALGORITHSION: FAST SEARCH

5.1 Introduction to Fast Search

After finding the best parameters for the ALS model, we may find some methods to speed up the recommending process at the query time. "Annoy" (Approximate Nearest Neighbors Oh Yeah) [2] is package that use a spatial data structure partition trees to accelerate search. As its name implies, the main idea of Annoy is to search for points in space that are close to a given query point. Ams in ALS, every user/item can be represented as a vector in high dimension space [2]. Their similarity can be calculated using some distance metrics. Annoy is a powerful because it has the ability to use static files as indexes and use less memory than other libraries [2].

In this experiment, we will try to compare and evaluate its performance to the brute-force method and research the efficiency-accuracy trade off by tuning hyper-parameters (tree size and search k).

5.2 Experiment setup

To start with, we firstly find the user matrix U and item factor V that produced by matrix factorization from best ALS model. We use the ALS model setting of rank = 50 and regParam = 0.5 (best hyperparameters in Section 4). Due to the high traffic in Peel cluster, we only use the 5% training dataset in this experiment. We also setup a brute-force method as our baseline model and its recommend results are treated as the ground truth labels. The performance of Annoy algorithm is measured by taking top 500 recommendations and measure if they are in the ground truth labels, which is also known as *recall*.

Among the distance metrics, we choose to use Dot Product distance because it is very common and easy to apply in brute force method. The choice of two main hyper-parameters are n_tree = [2, 5, 10, 20, 30, 50] and $search\_k$ = [10^2,10^3,10^3,10^4,10^5]. The detailed explanation will be introduced in the next section.

Table 2. Fast Search Time

Time(seconds)	k_trees = 2	k_trees = 5	k_trees = 10	k_trees = 20	k_trees = 30	k_trees = 50
search_k = 10^2	3.73^{-4}	3.66^{-4}	3.66^{-4}	3.80^{-4}	3.81^{-4}	4.33^{-4}
search_k = 10^3	1.58^{-3}	1.59^{-3}	1.53^{-3}	1.64^{-3}	1.55^{-3}	1.66^{-3}
search_k = 10^4	2.32^{-3}	2.33^{-3}	2.31^{-3}	2.29^{-3}	2.33^{-3}	2.44^{-3}
search_k = 10^5	8.26^{-3}	8.44^{-3}	8.69^{-3}	8.57^{-3}	8.72^{-3}	8.81^{-3}

Table 3. Fast Search Recall

Percentage	k_trees = 2	k_trees = 5	k_trees = 10	k_trees = 20	k_trees = 30	k_trees = 50
search_k = 10^2	0.037	0.043	0.052	0.057	0.060	0.061
search_k = 10^3	0.132	0.138	0.149	0.186	0.203	0.229
search_k = 10^4	0.215	0.296	0.338	0.365	0.376	0.391
search_k = 10^5	0.906	0.462	0.552	0.626	0.661	0.689

5.3 Experiment result and tradeoff

Our results are shown below, where x-axis measures the time efficiency (in seconds) and y-axis measures the accuracy by comparing Annoy results to brute-force results. In the image, a larger value in y-axis indicates better model performance and a smaller value in x-axis indicates better time efficiency.

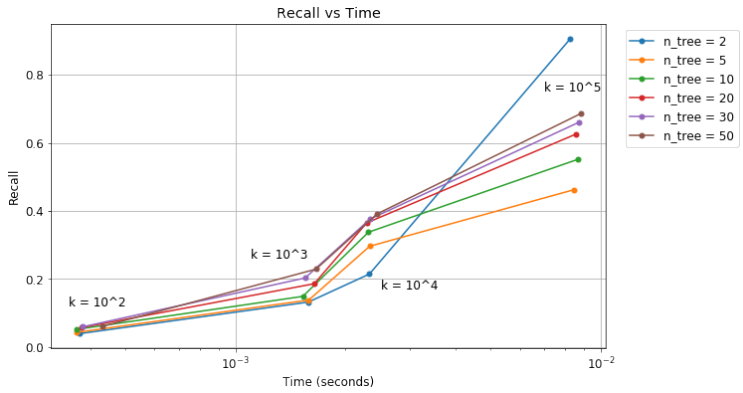

Fig. 2. The x-axis measures the average time (seconds) to generate recommendation and y-axis measures the accuracy

The graph shows that *recall* increases as *n_tree* increases and $n\_tree = 50$ gives the best accuracy. This result accords with "A larger value will give more accurate results, but larger indexes" given in Annoy website [2]. Time efficiency change is not obvious but lower *n_tree* are sightly more efficient in general. The other hyper-parameter *search_k* also affects the results, and a larger *search_k* value gives better accuracy but lower efficiency.

Based on the analysis above, there is an accuracy-time trade-off in this model. While implementing Annoy model, it is important to try different hyper-parameters and find a balance point between them. But in general, Annoy speed up the recommendation in a significantly scale, comparing to averaged 0.0158 seconds per user in brute-force method. Annoy can also generate recommendation with a very descent recall accuracy and we believe by choosing a larger range of hyper-parameters, Anooy will always be a great choice.

6 CONTRIBUTION

Ziyi Xie contributes mainly on ALS model and Letian Jiang contributes mainly on Fast Search.

REFERENCES

[1] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*.

[2] erikbern. [n.d.]. spotify/annoy. https://github.com/spotify/annoy

[3] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic Aspects in Information and Management*, Rudolf Fleischer and Jinhui Xu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–348.