Million Song Dataset Recommender System

JIANYU ZHANG, jianyu@nyu.edu
LIBEN CHEN, lc4438@nyu.edu
WENXIN ZHANG, wz2164@nyu.edu
SI GAO, sg6766@nyu.edu

Additional Key Words and Phrases: big data, spark, recommender system, collaborative filtering

ACM Reference Format:
Jianyu Zhang, Liben Chen, Wenxin Zhang, and Si Gao. 2021. Million Song Dataset Recommender System. 1, 1 (May 2021), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this project we build and evaluate a large-scale recommender system using Million Song Dataset (MSD) on Spark, Hadoop Distributed File System (HDFS) and single machines respectively. We apply collaborative filtering using ALS algorithm [3] on Spark to provide personalized music recommendation, and evaluate the performance of the recommender system measured by precision@k and mean average precision. We also implement two extensions on top of the baseline collaborative filtering model. The first extension implements a popularity-based baseline model as a benchmark; the second extension compares the performance and time costs with single-machine implementations (LightFM [4]).

2 DATA DESCRIPTION

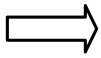
The Million Song Dataset (MSD) is collected by Bertin-Mahieux et al. [1], which consists of one million songs. The given dataset has already been divided into a training set, validation set and test set. Specifically, the training set includes both full histories of the 1 million users and partial histories of the 11 thousand users, while the validation and test set contain the remainder records of those 11 thousand users. Each row in the dataset represents an interaction between the user and the song, as well as the total counts that the user has played the song.

We employ PySpark to preprocess the data and build the model. Due to the limited computing resources, we first downsample the nearly 1 million users to 20 percent in the training set, and append them to those 11 thousand users who also appear in the validation and test sets. We tune parameters and evaluate test results based on the downsampled training set. Besides, we utilize StringIndexer to transform the columns into indices before feeding them to the ALS model to improve the model implementation efficiency. Figure 1 shows the process of our StringIndexer.

Authors' addresses: Jianyu Zhang, jianyu@nyu.edu; Liben Chen, lc4438@nyu.edu; Wenxin Zhang, wz2164@nyu.edu; Si Gao, sg6766@nyu.edu.

Manuscript submitted to ACM

	user_id	count	track_id
0	b80344d063b5ccb3212f76538f3d9e43d87dca9e	1	TRIQAUQ128F42435AD
1	b80344d063b5ccb3212f76538f3d9e43d87dca9e	1	TRIRLYL128F42539D1
2	b80344d063b5ccb3212f76538f3d9e43d87dca9e	2	TRMHBXZ128F4238406
3	b80344d063b5ccb3212f76538f3d9e43d87dca9e	1	TRYQMNI128F147C1C7
4	b80344d063b5ccb3212f76538f3d9e43d87dca9e	1	TRAHZNE128F9341B86



user_id	count	track_id
1	1	1
1	1	2
1	2	3
1	1	4
1	1	5

Fig. 1. A demo of StringIndexer

3 BASIC RECOMMENDER SYSTEM — ALS ON SPARK

3.1 Model Description

Alternating Least Squares (ALS) acts as an iterative optimization process. With each iteration we try to arrive closer to a factorized representation of our original data. The target of ALS is to decompose the sparse user-item interactive matrix M in to the product of two matrices U, V as:

$$M = U \times V \tag{1}$$

where U and V both have a low rank. The product of U, V can provide information about the empty entities in M, i.e. we want to use the estimated U, V to predict the ratings. This process is called matrix factorization. In theory, we would not be able to identify entities of M without constraints with matrix factorization since they can be any values. In practice, we assume the matrix M is low-rank, such that the low-rank matrix can be decomposed as in Equation 1.

ALS is an algorithm that can be largely paralleled. At each step, ALS freezes U or V and updates the other one. Furthermore, while we only need to update one matrix with the other one fixed, we update each row/column of the other matrix in parallel.

3.2 Evaluation Metrics

We mainly use two metrics to evaluate the performance of the recommender system.

- Precision@k: In this context of recommender system, precision metric is used to evaluate the binary output and indicates whether the user has listened to a specific track. Considering that we are most likely interested in recommending top-K songs to the user rather than all songs, we manually set K=500 to evaluate the fraction of relevant musics among the top-K retrieved songs.
- MAP: It happens that Mean Average Precision(MAP) is also useful to evaluate the recommender system. Using MAP to evaluate the recommender system implies that we are treating the recommendation like a ranking task. A user has a finite amount of time and attention, so we not only want to know just the 500 musics they might like, but also which one are most liked or which we have the most confidence in. MAP not only allows us to show the top recommendations, but also to market these top musics more aggressively.

3.3 Parameter Tuning

Considering the hyperparameters in our baseline model, 'RankParam' represents the number of latent factors; 'RegParam' specifies the regularization parameter; 'AlphaParam' is the parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations.

To find the best combinations of interacting hyperparameters, we perform a grid search over the following range of hyperparameters, and also plot the Precision at K metric in the figure below:

- RankParam ∈ [5, 10, 20]
- RegParam ∈ [0.1, 1, 10]
- AlphaParam ∈ [1, 5, 10]

3.4 Results

Figure 2 and Figure 3 show the performance of the baseline ALS model on the validation set based on Precision@500 and MAP respectively. It exhibits that when the RankParam = 20; Alpha = 10; RegParam = 0.1, the best performance is achieved within the validation set, where Precision@500 equals 0.0080, and M.A.P equals 0.0315.

Fig. 2. Precision@500 with different parameter settings

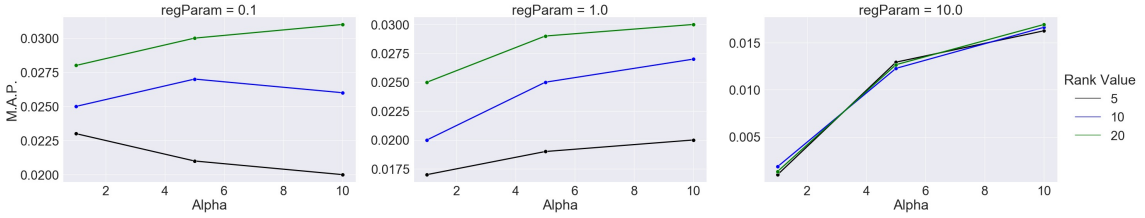


Fig. 3. M.A.P with different parameter settings

Therefore, we choose to evaluate our model on the test set using these parameters and summarize the results in Table 1:

Table 1. Performance of Baseline model on the test set

Model	MAP	Precision@500
Baseline Model	0.0315	0.0080

4 EXTENSIONS

4.1 Popularity-based Baseline Model

It is important to establish a reference point for the ALS model. In this section we discuss the commonly used baseline model that is built based upon popularity. This model does not involve user-item interactions and works in the principal

Table 2. Popularity-based Baseline Model

Load time (s)	Run time (s)	Precision@500	NDCG@500
35.32	2942.53	1.757×10^{-5}	2.45×10^{-4}

of popularity or trend. For the baseline model we not only consider the global popularity term but also model the user and item biases:

$$R_{u,i} = \mu + b_i + b_u \tag{2}$$

where μ is the global mean rating, b_i is the item bias, and b_u is the user bias. The predictions for user u are generated by simply sorting on the item bias b_i, since μ and b_u are both fixed.

We use *lenskit* package [2] to implement the bias-only popularity-based model. We apply the Bayesian damping at value 5 in order to stabilize rating estimates. Note that the original popularity model is established based on explicit user ratings. We assume that play counts can be used to indicate users' preferences for the items. This assumption suggests that we can use the implicit feedback as proxies for explicit ratings. We use the training set to estimate the global, user and item terms (μ, b_u and b_i), respectively. The model is estimated using 100% sample from training set.

Table 2 reports the baseline model evaluation results over the test set. Following the class instructions we use user rank = 500. The precision@500 is 1.757×10^{-5}, and normalized discounted cumulative gain at k is 2.45×10^{-4}.

4.2 Single Machine Implementations

Spark is powerful in dealing with Big Data tasks. However, it is more complex and expensive than single machine implementations. For the Million Song recommender system task, some single machine implementations are also capable, such as LightFM [4]. So it is an interesting task to compare time cost and performance of the Spark implementation and the single machine implementation (LightFM).

Indeed, the parallel ability of LightFM is up-bounded by the number of cores in a single machine. And the number of cores is not too large in most single machines (16 to 128 cores). While Spark can be largely paralleled due to the cluster property. We want to see if the capacity of a single machine is enough for the Million Song recommender system task in both time cost and recommendation performance directions.

4.2.1 LightFM Experiment. LightFM is an efficient implementation of Matrix Factorization which can map both user's music listening behaviors plus user's attributes and item's costumer records plus item's attributes to a shared latent space. Because of this property, LightFM can use both user and item's attributes easily. Furthermore, LightFM supports multi-thread which means we can parallel it on a single machine.

LightFM almost has no idea to stop overfitting and thus it is easy to overfit. A good way to solve this problem is validating on a validation set and using the earlystop trick. Earlystop trick is a common Machine Learning trick that stops training after we arrive at the best performance on a validation dataset. For all experiments of LightFM, we use a total of 20 threads executing in parallel.

For the single machine implementation, we search Rank $\in [10, 30, 100]$. Table 3 and Table 4 show the time costs and precision@500 performance of the LightFM single machine implementation with different Ranks. We can see that the time cost increases as the rank goes from 10 to 100.

It is interesting to compare the time cost and performance between the spark implementation on cluster and the LightFM implementation on a single machine. Comparing Table 3, 4 and Figure 2, we can see that the single machine

Table 3. Time Cost of Single Machine Implementation LightFM (20 threads)

Rank	10	30	100
Run time(s)	15120	18240	35220

Table 4. Performance (Precision@500) of Single Machine Implementation LightFM

Rank	10	30	100
Precision@500	5.2148×10^{-3}	5.2141×10^{-3}	5.2137×10^{-3}

implementation is slow in time and worse in precision@500 performance. Indeed, the main reason of performance comes from the choice of algorithms. The other reason of the time cost comes from the parallel scale. Spark on clusters can be largely paralleled.

5 CONCLUSION

In this project, we use the spark ALS algorithm to implement music recommendation on the large Million Song Dataset and evaluate the performance by precision@500 and MAP. After a grid search on parameters (rank, alpha and regularization) we find the best-performing hyperparameters and evaluate the recommendation performance over the test set. Furthermore, we also implement two additional extension algorithms: popularity-based baseline model with lenskit and a single machine implementation with LightFM. The goal of our first extension is to construct a simple but efficient baseline model. On the other hand, we want to compare the time cost and prediction performance between the spark ALS algorithm and the single machine implementation in the second extension. Finally we show that the spark implementation is fast in execution and has good prediction performance.

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] Michael D. Ekstrand. 2020. LensKit for Python: Next-Generation Software for Recommender Systems Experiments. In *Proceedings of the 29th ACM International Conference on Information amp; Knowledge Management* (Virtual Event, Ireland) *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 2999–3006. https://doi.org/10.1145/3340531.3412778

[3] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (Aug. 2009), 30–37. https://doi.org/10.1109/MC.2009.263

[4] Maciej Kula. 2015. Metadata Embeddings for User and Item Cold-start Recommendations. In *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015. (CEUR Workshop Proceedings, Vol. 1448)*, Toine Bogers and Marijn Koolen (Eds.). CEUR-WS.org, 14–21. http://ceur-ws.org/Vol-1448/paper4.pdf