Music Recommendation System

Yang Gao yg2410@nyu.edu
Xiaoying Huang xh2112@nyu.edu
Siyu Shen ss14359@nyu.edu
Yuqi Wei yw5260@nyu.edu

1. Introduction

The goal of this project is to build a collaborative-filter based recommender system using the Million Song Dataset and the user interaction dataset from the Million Song Dataset Challenge. The collaborative filtering method assumes that people will listen to similar songs that they have listened to before and it makes recommendations based on the play count information on the songs. In this project, we used the collaborative filtering method with the Alternative Least Square model from PySpark's built-in library to learn latent factor representations and conduct hyper-parameter tuning to improve the baseline and guard our model against overfitting. We further explored two extensions: Single Machine Implementation using the LightFM library and Fast Search for queries with the Annoy library to compare their performances with the baseline model.

2. Data

2.1. Dataset Description

The user-music interaction dataset comes from the Million Song Dataset Challenge, which consists of implicit feedback, play count, for approximately one million users. The dataset contains three columns: "user_id", "track_id", and "count", which indicates the number of times a user listened to a specific track.

2.2. Downsampling

Due to the limitation of computing resources, we downsampled the original dataset to 1%, 10%, and 25%. We used two methods for downsampling: the first method was to downsample and create a random sample from the whole training set. For the 25% sample, it contained 92% of the user information from the validation set. The second method was to select all training records for users that showed up in both the training set and the validation set.

2.3. StringIndexer

Because the columns "user_id" and "track_id" are strings and the ALS model requires them to be integers, we used StringIndexer from PySpark to convert the two columns into indices before fitting into the ALS model.

3. Model

We used the Alternating Least Square (ALS) model from the built-in library in PySpark. The ALS algorithm factorizes the utility matrix into user matrix U and item matrix V. By fixing one of the matrices U or V, it tries to uncover the latent factors of users and items by solving the optimization problem that is equivalent to ordinary least squares regression [Spark, a]. The ALS algorithm is also designed to run in parallel, which significantly improves efficiency.

4. Evaluation

We used three evaluation metrics for ranking to measure the model performance, which were MAP, Precision at 500, and NDCG. We mainly used MAP for hyper-parameter tuning and we evaluated the best-performing model using all three metrics. A brief description of the three metrics is shown below [Spark, b]:

- **Mean Average Precision (MAP)**
  It measures the number of predicted items that are in the true relevant item set, averaged across all users. It gives higher penalties for the items that are highly relevant but are not highly recommended by the model.

- **Precision at k**
  It measures the number of the top k recommended items that are in the true relevant item set, averaged across all users. The order of recommendation is not relevant. For this project, we used k = 500 because we evaluated the model based on the top 500 recommended items.

- **Normalized Discounted Cumulative Gain (NDCG)**
  It measures the number of the top k recommended items that are in the true relevant item set, averaged across all users. Unlike Precision at k, NDCG takes into account the order of recommendations.

5. Experiments and Results

5.1. Hyperparameter Tuning

Alpha = 10				
Rank	Reg	MAP	Precision k	NDCG
100	0.1	0.03999	0.008424	0.156926
	1	0.04010	0.008435	0.158148
150	0.1	0.042131	0.008593	0.161221
	1	0.042791	0.008611	0.161891
200	0.1	0.044331	0.008836	0.16636
	1	0.044801	0.008831	0.166799

Table 1: Hyperparameter Tuning Results

We applied grid-search for tuning three hyperparameters: rank (the number of latent factors in the model), regularization parameter (the regularization strength in ALS), and alpha (the baseline confidence in preference observations). We used MAP as the metric for tuning hyperparameters for data sets with sizes 1%, 10%, and 25%. The results of 25% train are shown in Figure 1. The general trends show that MAP increases with rank. We can see that as rank becomes larger, the differences in MAP between alpha = 10 and alpha = 1 become more prominent. We also find that the model performance with regularization parameter = 1 is better than that with 0.1 holding alpha constant at 10. Hence, the hyper-parameters combination we chose was (rank = 200, reg = 1, alpha = 10). We did attempt to raise rank to 300, but the increase in training time made the effort not worthy. To further justify our choice, we also compared Precision at 500 and NDCG at 500 for large rank values, and (rank = 200, reg = 1, alpha = 10) proved to be the best performing combination.

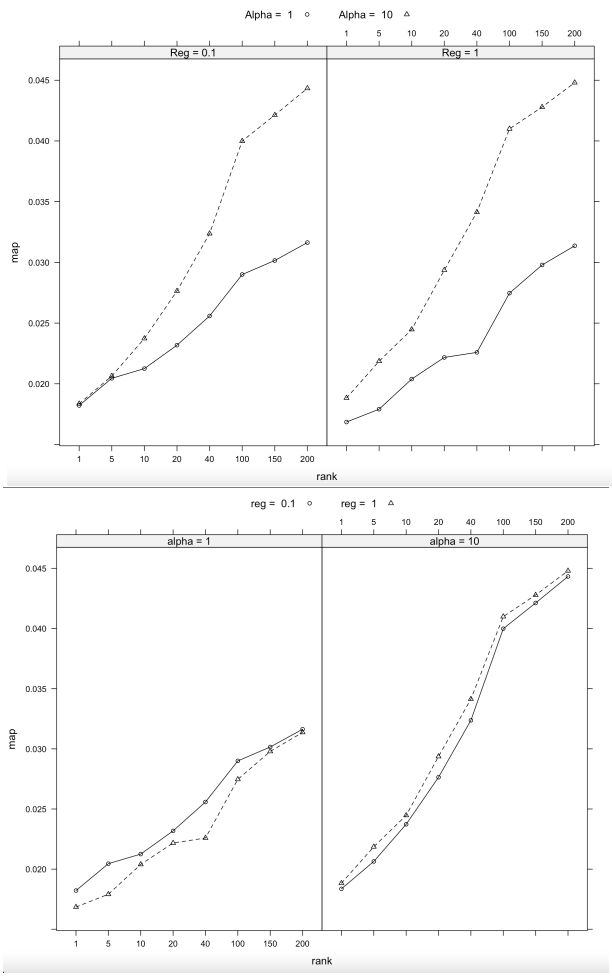

Figure 1: Comparison graphs for Alpha Differences (above) and Regularization Differences (below)

5.2. Model Performance on Test Set

After hyper-parameter tuning, we used the above selection to make predictions on the test set after training from the full training set. The MAP score on the test set is 0.0698 and the Precision at 500 is 0.01275.

6. Extensions

6.1. Single Machine Implementation with LightFM

LightFM [Kula, 2015] is a Python package for single machine implementation of a variety of recommen-

dation systems. Same as ALS, the model is designed for both implicit and explicit feedback. The author of the package claims that the LightFM model is "fast and able to produce high quality results" [Kula, 2015]. We therefore would like to compare both the accuracy and the efficiency of the LightFM model with the ALS model implemented in PySpark.

We trained a LightFM model on 1%, 2% and 10% of the training dataset to evaluate how the precision and speed vary as the size of the dataset increases. Since we did not train the model on the full training set, we encountered the 'cold start' problem common in recommendation systems. That is, there were new users and new tracks present in the validation set. When implementing the baseline ALS model, we set the 'coldStartStrategy' parameter to 'drop'; in order to be consistent, when fitting LightFM, we also dropped the users and tracks in the validation set that did not appear in the three training sets.

Our hyperparameters included the learning rate, which we tuned over [0.1, 1, 10], and the loss, which we tuned over 'BRP' and 'WARP'. 'BPR', abbreviated for 'Bayesian Personalised Ranking' loss, maximizes the prediction difference between a positive example and a randomly chosen negative example [Rendle et al., 2012]. 'WARP' (Weighted Approximate-Rank Pairwise) loss, on the other hand, maximises the rank of positive examples by repeatedly sampling negative examples until rank violating one is found [Weston et al., 2011]. Since the ALS model does not have corresponding hyperparameters, we tuned 'rank' over [1,5,10] and the regularization parameter over [0.1, 1] to also generate 6 combinations of hyperparameters.

Since we have calculated 'precision at k' in our baseline model, and it is also a built-in metric in the LightFM package, we used this metric for model comparison between LightFM and ALS. Table 2 shows the precision of the best parameter setting as well as the time taken to train and fit the models for LightFM and ALS using 1%, 2% and 10% of the training data, respectively.

As the results suggest, there is a trade-off between the efficiency and accuracy of the algorithms. The precision of ALS under the best parameter setting always outperforms that of LightFM although it also takes significantly more time to train. However, it is also important to note that the training and execution time of LightFM scales almost linearly with the amount of data; whereas for ALS, although there is also a positive relationship between training data size and time, the latter increases at a lower rate compared to LightFM.

Precision	Time	Model	% of Training
0.003487	187.3855	ALS	1%
0.004419	281.3126	ALS	2%
0.005829	571.6147	ALS	10%
0.003443	23.3680	LightFM	1%
0.004008	53.6038	LightFM	2%
0.005137	279.6613	LightFM	10%

Table 2: Precision and Speed of LightFM and ALS

6.2. Fast Search with Annoy

In practice, it is important for a recommender system to not only provide personalized recommendations that are accurate and generalizable, but also respond to query in a time efficient way. The Annoy library [Annoy] used at Spotify is one of the best so far to accelerate query time. The idea is that Annoy uses the random projection method of LSH to randomly choose a hyperplane at each node and repeats it k times to form a forest of trees [Wiki]. There are two hyper-parameters to tune: n_trees and search_k. If without memory or time constraints, the two parameters are advised to be set as high as possible for high precision. This is not the case in practice. Hence, we provide an efficiency gain analysis of Annoy vs. a brute force search method.

Due to limited cluster usage, we set up a local Spark environment and downloaded data to local computers to complete this extension. For better comparison with the Annoy method that uses dot product, the brute force method also computes the dot product of each input query and each item. The features here are extracted from our best model with rank = 200, reg = 1, and alpha = 10 using down-sampled data. We asked both methods to generate 500 recommendations for each input user and the brute force results are treated as ground-truth for precision calculation.

The plot in Figure 2 shows the relationship between efficiency gain and precision. We define efficiency gain to be: the ratio of mean run-time of a query search using brute force over that of Annoy.

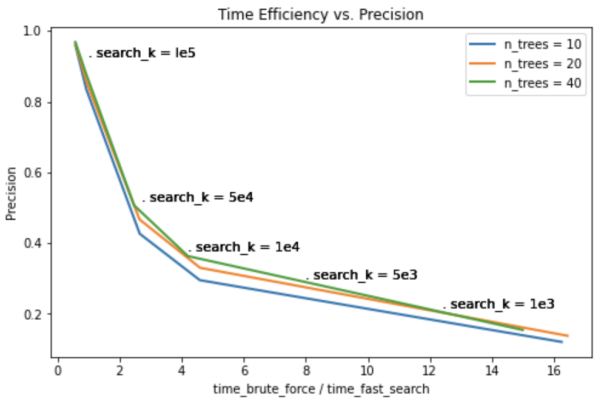

Figure 2: Annoy Plot with Time Efficiency vs. Precision

The lower search_k is, the faster the nearest neighbor search using Annoy, and the more time efficiency gain we have compared to brute force search. However, there is a trade-off between efficiency gain and precision. When search_k and n_trees are infinitely large, the results of approximated nearest neighbor search approach the ground-truth, but we fail to accelerate query time; when search_k is around 1000, the mean query time is only 1/10 of the brute force search time (see fast_search.ipynb), but we only achieve a precision of 0.2. A search_k of around 8000 and n_trees = 40 balances search time and precision, which is around 0.7. To conclude, such trade-off should be carefully considered when we use accelerated search libraries such as Annoy in practice.

7. Discussion

Due to time constraints, we tuned the ALS model mainly with the 25% training set instead of the full set, but we did run our best parameter combo (rank = 200, reg = 1, alpha = 10) on the full training set and used that to predict validation (MAP = 0.0695) and test (MAP = 0.0698). If time and resources allow, we would be interested in searching for the threshold in rank above which the MAP score would start to fall. Also, when doing the LightFM extension, we realized that the time comparison may not be very rigorous, since running ALS on Peel can be slow as 80 other jobs are active at the same time, but this is not a big problem for running LightFM on Greene.

8. Contribution and Collaborations

- Xiaoying Huang: Baseline, Extension 1, Report
- Siyu Shen: Baseline, Extension 1, Report
- Yang Gao: Baseline, Extension 2, Report
- Yuqi Wei: Baseline, Extension 2, Report

References

Apache Spark. Collaborative filtering, a. URL https://spark.apache.org/docs/2.4.7/ml-collaborative-filtering.htmls.

Apache Spark. Evaluation metrics - rdd-based api, b. URL https://spark.apache.org/docs/2.4.7/mllib-evaluation-metrics.html#ranking-systems.

Maciej Kula. Metadata embeddings for user and item cold-start recommendations. In Toine Bogers and Marijn Koolen, editors, *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.*, volume 1448 of *CEUR Workshop Proceedings*, pages 14–21. CEUR-WS.org, 2015. URL http://ceur-ws.org/Vol-1448/paper4.pdf.

Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618*, 2012.

Jason Weston, Samy Bengio, and Nicolas Usunier. Wsabie: Scaling up to large vocabulary image annotation. 2011.

Annoy. Github repository. URL https://github.com/spotify/annoy.

Wiki. Locality-sensitive hashing. URL https://en.wikipedia.org/wiki/Locality-sensitive_hashing#Random_projection.