Recommender System on the Million Song Dataset

Introduction

In designing our recommender system based on the latent factors of the Million Song Dataset our group proceeded through the main stages of parallel system deployment, and arrived at a functioning and optimized ALS model. These stages are outlined below as Preprocessing, Downsampling, Training and Evaluation, and Hyperparameter Tuning. We primarily based our decision making throughout the process on mean average precision, and then evaluated final products on further metrics, such as NDCG and precision at 500. We then extended our modeling to include a comparison to a popularity-based baseline model, as well as a local implementation of LightFM. Our parallel ALS model outperformed on all metrics.

Preprocessing

The original Million Song Dataset only contains string format user and track index, but the Pyspark API only accepts integer format index to fit the model, so our goal in the preprocessing step is to assign a unique integer index to the original string index that is consistent to three parquet files. To do that, we created two huge mapping data frames for both users and tracks from a unioned data frame consisting of train, validation and test data. Then two mapping data frames inner joined with three individual data frames. Finally we dropped the original indices from them, repartitioned the data by new user and track index and wrote the HDFS disk for future use.

Downsampling

In order to develop our model and iterate more rapidly we sub-sampled the data. We created three separate sample sizes of the data: 1) small (1%), 2) medium (5%), and 3) large (25%). To do so, we created a subset of users from the union of all the users across all three datasets (train, validation, and test). We then filtered each of the three datasets by that sample of users. This ensured we had enough overlap of users in all three datasets when downsampling. Therefore, we avoided the problem where a user is in the validation set but not the training set as well as the inverse problem where the user is in the training set but not the validation set.

Training & Evaluation

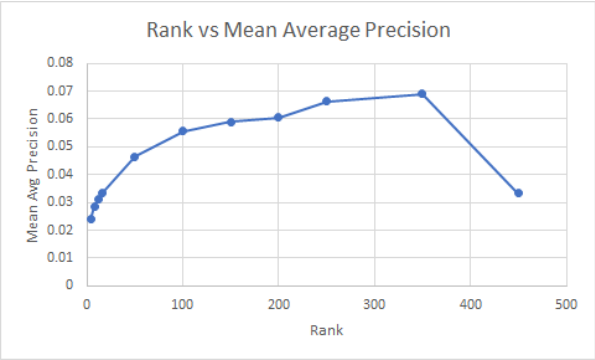
The training part for Alternating least square model is straightforward. We just create an ALS model instance from the Pyspark recommendation package, setting the parameters and fit on the training data frame. However, the evaluation step is a bit tricky as the Pyspark's RankingMetrics API requires a specific RDD format to evaluate, which contains prediction and true label pairs. To create ground truth labels, we add a new column that contains a list of track indices ordered by their corresponding counts of each user for the data frame being evaluated. Then we make a prediction data frame using the RecommendForUser function from the ALS model, which could predict top items on each user in evaluated data using the model we just fit. We then join the two data frames by user index and transform into RDD that is valid for the ranking metrics function to calculate any ranking evaluation metrics we need. We finally used the optimal hyperparameter set to train the whole training set and evaluate on the testing set.

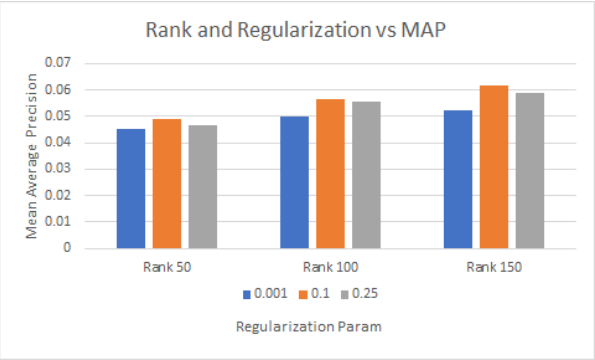
Table 1: Full Test Result Summary

Mean Average Precision	Precision at 500	NDCG at 500	Time cost / s
0.065434	0.012443	0.23519	28825.23

Hyperparameter Tuning

The parameter tuning process was fairly straightforward, but dependent on the restrictions of running time and the congestion of the cluster during this phase. We began by performing a grid search on our downsampled small file. We initially used relatively low ranks and varied the regularization parameter in intervals between .001 and 2. We found that the overall mean average precision improved up to the regularization equalling .25. Between .25 and 1 the precision cost was very moderate, so we were left with a somewhat subjective decision on balancing generalization vs training precision. By 2 the model precision had fallen dramatically. We also varied maximum iterations between the default 10 up to 30, finding optimal results at 20. We progressively tested these settings on the larger files to ensure the pattern remained consistent, which it did. We then scaled up to significantly higher ranks, finding our best results at a rank of 350, with performance falling significantly by 450.





Extension 1: Popularity-Based Baseline Model

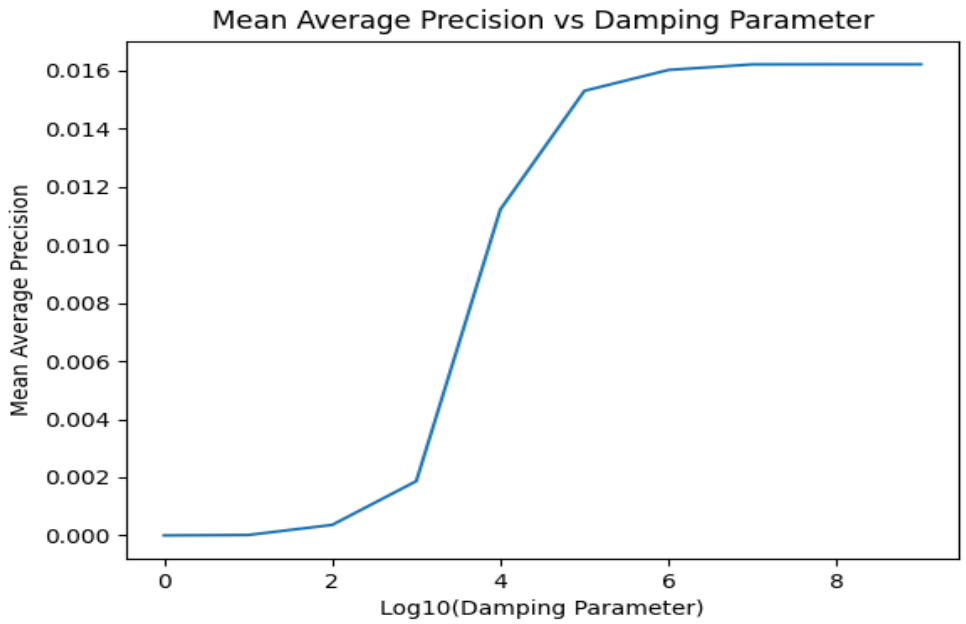
For extension 1, we chose to implement our own popularity-based baseline model. It is always important to start simple and have a baseline comparison. In fact, often the simple popularity model of the items is difficult to beat. And once we have implemented such a model, we can then compare its performance to our ALS model to determine how much improvement we gain by modeling interactions directly and personalizing the model.

Spark does not provide a popularity model, so we used the bias implementation provided by LensKit. The only tuning we had to perform was that of the damping parameter. The damping parameter, also referred to as a pseudo count, regresses the estimates of ratings toward zero. Items with a lot of interactions are not affected as much as items with very few interactions. This ensures that the ratings are not unstable due to very few interactions. We tuned our damping parameter on the training and validation set. We

assigned ten different values (ranging from 0 to 9) increasing geometrically by a common ratio of 10. As seen in the plot below, we found that a damping parameter of a billion (10^9) gave us the best results. Such a large damping parameter decreases the ratings of tracks that have high counts but few listeners. All that matters with a large damping parameter is the total number of plays across all users, as it is approximately proportional to the total number of counts instead of the average number of counts per user. After parameter tuning, the model was trained on the training and validation sets combined, then evaluated on the test set, using Spark's ranking metrics . The major technical issue we ran into was generating ranking metrics consistent with Spark. To produce these ranking metrics, we used SparkContext's parallelize method to convert our dataframes to RDDs to work within the spark framework. The mean average precision on the full test set can be found in table 2**.** After tuning the damping parameter to optimize mean average precision, we also computed precision and NDCG at k=50. The appropriate value of k might depend on business decisions and how many recommendations a company wants to display.

Table 2: Full Test Result Summary

Mean Average Precision	Precision at 50	NDCG at 50
0.017674	0.012534	0.044461



Extension 2: Single Machine Implementation

In this section, we have used the lightfm package for recommendation system implementation in Python and compared our experiment results. We also utilized Pandas pivot table and SciPy csr_matrix in intermediate steps for data structure transformation.

Implementation with LightFM

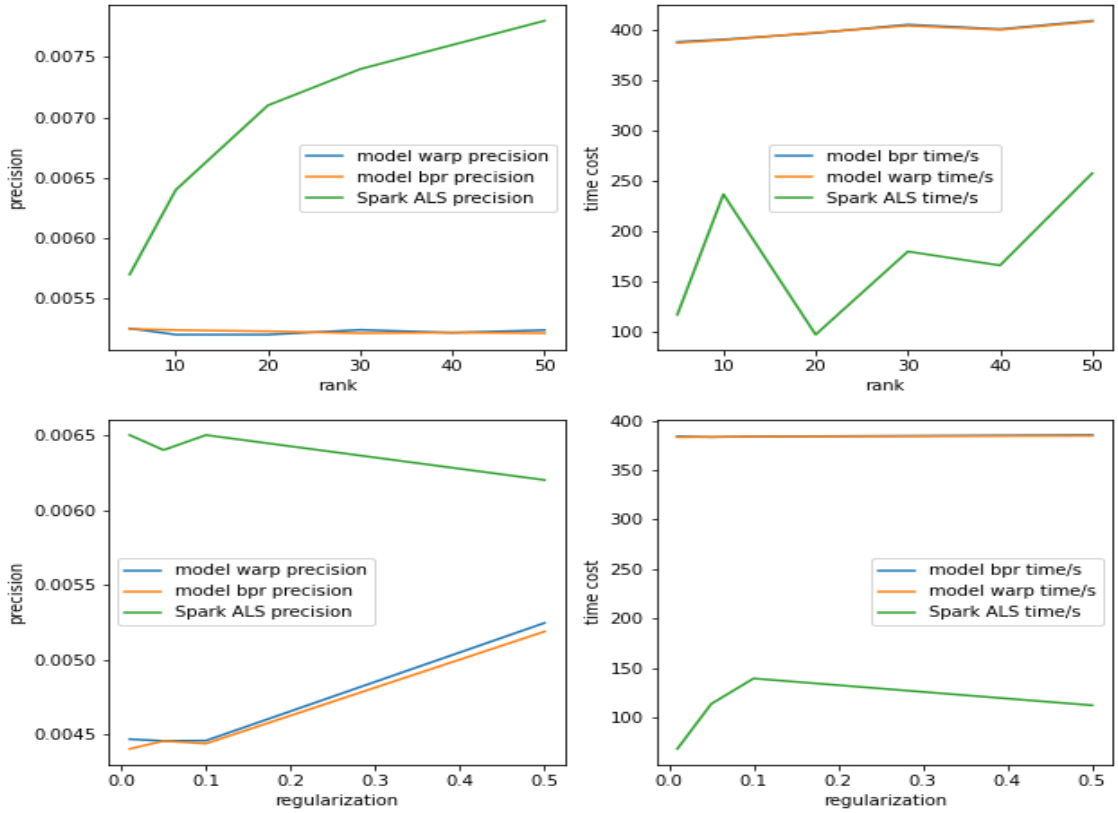
Lightfm is a package designed for various types of recommender systems. It covers functions for both implicit and explicit feedback models, and can be applied to collaborative filtering models, hybrid models as well as cold-start recommendations. It is used for both implicit and explicit feedback, including efficient implementation of BPR and WARP ranking losses. It's easy to use, fast (via multithreaded model estimation), and produces high quality results.
It also makes it possible to incorporate both item and user metadata into the traditional matrix factorization algorithms. It represents each user and item as the sum of the latent representations of their features, thus allowing recommendations to generalize to new items (via item features) and to new users (via user features).

In this project we only explored the collaborative filtering part of the lightfm package. A summary of implementation steps is listed as follows:

• Input dataframe of interactions, convert to initial utility matrix with pandas.pivot_table, then convert to coordinate format with scipy.sparse.crs_matrix for fitting lightfm models.
• Training and test split with built-in function lightfm.cross_validation. We split into train and test sets, corresponding with test and train sets used for Spark ALS model.
• Fit dataset into a lightfm instance. We conducted experiments on both Weighted Approximate Rank Pairwise Loss (WARP) and Bayesian Personalised Ranking (BRP) for better performance comparison.
• Get test score of precision at k = 500. The total time consumed for fitting and testing the model was also recorded.

Performance comparison with Spark's ALS model

We ran lightfm model (both warp and bpr) on 1% and 5% dataset, with the same hyperparameter combinations as in Spark ALS. Some instant observations are:
• Spark ALS and LightFM produce roughly the same magnitude of ranking metrics and evaluation time, although in different patterns. There's only minor differences between lightfm warp and bpr methods.
• The optimal hyperparameter combinations for Spark ALS and LightFM are different. This may result from different splitting methodologies of the original dataset. Comparisons between lightfm and Spark ALS, on precision at k and time, over ranks and regularization parameter. Comparisons between lightfm warp, bpr models and ALS model, on precision at k and time, over ranks and regularization parameter is shown below.



Finally, it's worth mentioning that lightfm has a limitation for large scales of datasets. The transforming step to pandas pivot table and scipy sparse csr matrix both contain dimension restrictions. Also lightfm only contains limited built-in evaluation metrics impacting the choice of tools to use for other systems.

Contributions

Zixuan Shao: Preprocessing, Train & Evaluation, Assisting LightFM implementation

Jack Jones: Downsampling, Extension 1: Popularity-Based Baseline Model

Prerna Mishra: Extension 2: Single Machine Implementation using LightFM

Zach Vogel: Hyperparameter Tuning