Implementing a Spark ALS Recommender System for the Million-Song Dataset

Jordan Chervin, Priyanka Shishodia
DS-GA 1004: Big Data, S'21

Introduction

Recommender systems are everywhere: our weekly Discover playlist on Spotify, the "related to items you've viewed" section of products on Amazon's home page, "shows you might like" on Netflix, and more. Fundamentally, recommender systems are algorithms that predict a relevant set of items for users based on their past history, their similarity to other users, or explicit feedback.

In this project, we will use collaborative filtering to implement a basic recommender system using Spark's Alternating Least Squares Method to learn latent factor representations for users and tracks, and predict the top 500 songs for each user. Collaborative filtering aims to fill in missing entries of a sparse user-item matrix. In the alternating least squares method, the model alternates between training over the users and items. The ALS module is advantageous for this project because of its scalability and parallelizability.

We will also explore other baseline models such as popularity-based and user-item bias rating algorithms.

Model Implementation and Method

The data for this project comes from the Million-Song Dataset, a collection of audio features and metadata for one million contemporary popular music tracks. The primary dataset consists of tuples in the form of `(user_id, count, track_id)`, where "count" is the number of times a user listened to a particular track, representing implicit feedback. Because `user_id` and `track_id` are alphanumeric strings and the ALS module only supports integers, we used PySpark's StringIndexer method to create unique integer indices for each user and track.

We initially used RDD mapping to convert the transformed data from the previous step into the format needed; however, training on 1% of the training set took hours longer than we expected with maxIter=5, regParam=0.01, implicitPrefs=True, coldStartStrategy="drop", and default rank and numBlocks. Local troubleshooting on Colab revealed that we were getting stuck on `model = als.fit(train_df)` because too many delayed RDD computations were happening at this step. Therefore, we optimized this step by employing PySpark Dataframes instead of RDDs.

Our baseline ALS model ultimately used the following parameters:

- maxIter = 5
- numBlocks = 10 (default)
- rank = 10 (default)
- regParam = 1 (default)
- implicitPrefs = True
- coldStartStrategy = “drop”

Next, we performed hyperparameter tuning over the following rank and regularization parameters, and evaluated these models using RMSE to inform further hyperparameter tuning and evaluation using ranking metrics. Hyperparameter tuning over rank is important because rank represents the number of latent factors in the model, and provides control on expressivity and complexity.

- rank = [10, 50, 100, 150]
- regParam = [0.01, 0.1, 1, 1.5]

Finally, based on the RMSE scores from the first 16 models, we did additional hyperparameter tuning over the following rank and regularization parameters.

- rank = [8, 10]
- regParam = [0.01, 0.1, 0.3, 0.5]

However, similar to the earlier problem of having too many delayed RDD computations when fitting, many of our hyperparameter-tuning jobs were killed when the script reached `metrics = RankingMetrics(predictionAndLabels_rdd)`, which we believe was for a similar reason that we could not resolve.

Evaluation and Results

There are two classes of metrics that can be used to evaluate our recommender system: regression metrics and ranking metrics. The regression metric we initially chose to evaluate our models during hyperparameter tuning was root mean squared error (RMSE), whose mathematical formula is:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=0}^{N-1} (y_i - \hat{y}_i)^2}{N}}$$

Because the RMSE calculation does not require any RDD computation in PySpark, we were able to successfully evaluate the full validation set trained on both 10% of the training set (Appendix A) and then on the entire training dataset (Appendix B), relatively quickly over the aforementioned 16 different rank and regularization parameter combinations. The combination that yielded the smallest RMSE (7.690148) was rank=10 and regParam=0.1. Of note, RMSE values for the validation set trained on the whole training set only marginally improved after increasing the size of the training set from 10% to 100%.

However, ranking metrics are more meaningful in the context of recommender systems, because the output is an ordered list of the predicted most-relevant items to a user. The three ranking metrics we are using to evaluate our model are mean average precision, precision at k, and normalized discounted cumulative gain. Mean average precision is a measure of how many of the recommended items are in the set of true items, where order matters. Precision at k is a measure of how many of the first k recommended items are in the set of true relevant items, averaged across all users. Finally, normalized discounted cumulative gain is the same as precision at k; however, NDCG at k takes order into account, whereas precision at k does not. For this project, k = 500.

We encountered some complications during this project which mostly arose from competing for resources on the cluster and attempting to debug thorny error messages involving memory. These frustrations forced us to only use 10% of the training set when fitting the models that we were evaluating using ranking metrics, in order to get *some* results. Therefore, we were not able to tune our parameters as finely as we would have preferred, and our scores are lower than anticipated. We hypothesize that they would marginally improve with training on the full dataset, and perhaps more finely-tuned hyperparameters. Tables of our results on the validation set are below, and plots of these results can be found in Appendix C.

Rank=10

regParam	MAP	Precision at k	NDCG
0.01	0.011550	0.004203	0.071372
0.1	0.011384	0.004236	0.071448
0.3	0.009713	0.004110	0.067348
0.5	0.009446	0.004120	0.066723

Rank=8

regParam	MAP	Precision at k	NDCG
0.01	0.011701	0.004025	0.069791
0.1	0.010532	0.004315	0.070926
0.3	0.007993	0.004015	0.062914
0.5	0.007620	0.003882	0.060320

The parameter combination that gave us our best results on the validation data (based on what limited tuning we could perform) was rank=8 and regParam=0.01. RMSE on the test set using these parameters was 0.988383. We were not able to evaluate the test set using ranking metrics for the reasons described earlier, although we expect the scores of the evaluation metrics on the test set to be close to the validation ones.

Extension

We implemented two alternative baseline models using LensKit, a Python tool for experimenting with and studying recommender algorithms:

a) Bias Model

The user-item bias rating algorithm was trained on the same pre-processed training dataset. The algorithm implements the predictor algorithm that is a sum of the global mean rating, item bias and user bias. The trained model is evaluated on the entire validation set using RMSE, per-user RMSE and Mean Average Error (MAE). The following table depicts the results.

	RMSE	Per-user RMSE	MAE
Bias Model	7.5897	4.7535	3.1644

b) Popularity Model

Popularity-based recommendation is a successful non-personalized method that is based on items that are popular or trendy at a given time. We implemented the most-popular-item algorithm from LensKit on the same pre-processed sample dataset. In order to map the track index back to the original string IDs, PySpark's decoder IndexToString() was used. The following table depicts the track recommendations and popularity score (subject to the 10% sample training set).

Rank	Popularity Score	Track ID	Rank	Popularity Score	Track ID
1	1126.0	TRDMBIJ128F4290431	6	830.0	TRONYHY128F92C9D11
2	930.0	TRGXQES128F42BA5EB	7	717.0	TRLGMFJ128F4217DBE
3	913.0	TRAEHHJ12903CF492F	8	700.0	TRVCUSW128F92F20C6
4	866.0	TRHKJNX12903CEFCDF	9	676.0	TRVSBTV12903CC6670
5	846.0	TROAQBZ128F9326213	10	585.0	TRTNDNE128F1486812

References

1. https://towardsdatascience.com/build-recommendation-system-with-pyspark-using-alternating-least-squares-als-matrix-factorisation-ebe1ad2e7679
2. https://spark.apache.org/docs/2.4.7/ml-collaborative-filtering.html
3. https://spark.apache.org/docs/2.4.7/mllib-evaluation-metrics.html#ranking-systems

Appendix

A.

RMSE scores of the full validation set on 10% of the training set

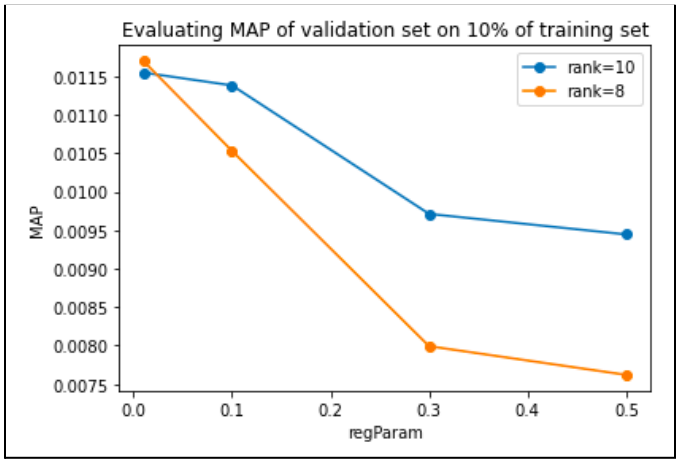
rank	regParam=0.01	regParam=0.1	regParam=1	regParam=1.5
10	7.712670	7.712096	7.713621	7.714108
50	7.712389	7.712270	7.713528	7.713927
100	7.712249	7.712296	7.713484	7.713874
150	7.712235	7.712258	7.713471	7.713889

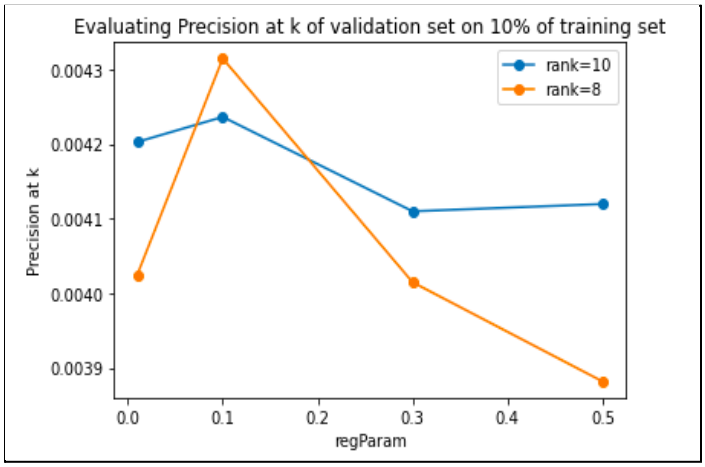
B.

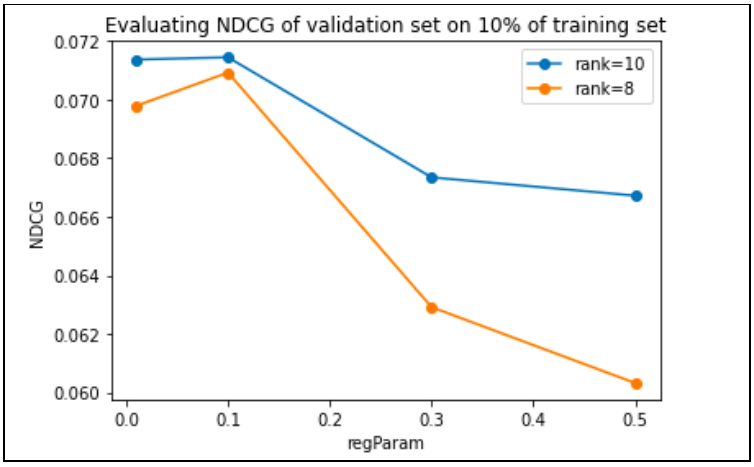
RMSE scores of the full validation set on the full training set

rank	regParam=0.01	regParam=0.1	regParam=1	regParam=1.5
10	7.691884	7.690148	7.705564	7.709698
50	7.693255	7.692822	7.705322	7.709374
100	7.693947	7.693768	7.705202	7.709190
150	7.694136	7.694049	7.705205	7.709169

C. Plots of our evaluation metrics of the validation set on 10% of the training set:







D. Group member contributions:

Both team members contributed equally to the final report write-up. Priyanka took the lead on data pre-processing, training set sub-sampling, and the extension. Jordan took the lead on hyperparameter tuning, evaluation, and the tables and plots. We both troubleshooted together where necessary. More detailed and granular contributions can be found in our git commit history.