Collaborative Filtering using Spark's ALS Model

KIERAN SIM, JACK EPSTEIN, PEDRO GALARZA, and SARA PRICE

1 INTRODUCTION

Our goal is to build a collaborative filtering model that generates ranked lists of 500 song recommendations for users in the Million Song Dataset (MSD). The MSD contains records of play counts by user by track as well as metadata for tracks. Given the size of the MSD with close to 50 million records overall, 385,371 unique tracks, and 1.13 million unique users, we use Spark and distributed computing for implementation. We train and tune models using the validation dataset of 10,000 unique users and are able to achieve a Mean Average Precision (MAP) of 8.3% on the test dataset of 100,000 users.

2 BASIC RECOMMENDER SYSTEM

Our approach is a collaborative filtering model built using PySpark's Alternating Least Squares (ALS) model. We treat the play count data as implicit feedback, which means the model uses the framework laid out in [1] and treats play counts as user preferences. We control the weight of these preferences in the model through hyperparameter tuning (see Section 2.3). The ALS model generates and optimizes matrices representing sets of latent factors for each user and each item. To make recommendations for users, it computes similarity scores between each user and each track and then rank orders tracks from most to least similar. We generate the 500 song recommendations per user using ALS.recommendForUserSubset().

For model evaluation, we primarily use Mean Average Precision (MAP) because, unlike other metrics like Root Mean Square Error (RMSE), MAP rewards the model for recommending high confidence predictions first. Therefore, it is not just evaluating overall metrics like True Positive Rate but how effectively the model has ordered its recommendations. We drop predictions for users that are not in our validation set.

2.1 Data Preparation

The raw data set has the following relevant schema: {userId str, itemId str, count int}. The ALS model requires userID and itemID to be integers, so we convert these columns using the default pyspark.sql.functions hash function, which uses MurmurHash. For downsampling purposes, we also generate flags indicating which users are in the validation or test sets. As detailed in the following section, when downsampling the training set, we make sure to keep all data associated with users in the validation set. This is analogous to a real-world scenario in which we are providing recommendations for a pre-known list of users that have strategic value to us.

For model evaluation, we aggregate and sort play counts for users in the validation set to use as proxies for true user preferences against which to score our recommendations. None of the users in the validation data set had listened to 500 distinct songs, so our ground truth for each user is capped at the number of songs we had available. Since we test many model iterations, we save this collected version of our validation user preferences to be reused.

2.2 Downsampling Methods

To get the pipeline of model training, prediction, and evaluation working on the cluster, we began by training a model on just users that are in the validation set, which is only 2.42 million records. This was a much more reasonably-sized dataset on which to perform debugging. Once the pipeline was running, we trained models using the full training dataset with different ranks and found an average runtime of about 1.25 hours for rank = 50, 2.5 hours for rank = 100, and 8.5 hours for rank = 200. We note that this was without any additional parallelism, which we incorporate into our process later.

Authors' address: Kieran Sim; Jack Epstein; Pedro Galarza; Sara Price.

These runtimes would have made extensive hyperparameter tuning challenging, so to speed up performance we test different approaches for dowsampling the training dataset. As noted previously, these approaches always include all training data associated with users in the validation set. Downsampling is only applied to the remaining 47.6 million records in the training data. The first approach we tried was a random downsample. The second was to downsample based on song popularity. We define popularity based on which songs have the most interactions with the most users. As shown in Figure 1 of Appendix A, this data has a long tail with about 90% of interactions occurring with less than one third of songs. Therefore, we hypothesize that biasing our sample towards popular items could actually improve model performance by skewing recommendations towards more popular items. Table 1 in Appendix A includes the results of these downsample methods using rank = 100, alpha = 5, maxIter = 15, and regParam = 0.1 on 10% and 25% downsamples. We find that 10% is too aggressive and results in a decrease in performance that is not worth the relatively small decrease in runtime. Somewhat surprisingly, the popular item downsample seriously under-performs the random downsample. In retrospect, this may be because, by definition, the random downsample does not significantly change complexity of the training data while the popular item downsample does. This loss of complexity could subsequently degrade quality of the learned latent factors. Inspecting the Learning Curve in Figure 3 Appendix A, we can see that a 25% downsample significantly decreases the training data size while not sacrificing too much in model performance.

2.3 Hyperparameter Tuning

Based on the downsample results detailed above, we pursue hyperparmeter tuning using the 25% random downsampled data. To facilitate this process we utilize ALS.fitMultiple(), which allows efficient training and fitting of multiple combinations of hyperparameters without extensive looping. To further speed up tuning we often ran multiple jobs in parallel and updated the SparkSession config to increase parallelism (see *code/models/create_models_and_predictions_multi.py* [222-229] in the GitHub repo). We also found that setting a specific checkpoint directory allowed us to experiment with higher maxIter values.

Table 2 in Appendix B details the ranges of hyperparameters tested. Tables 3 - 6 in Appendix B highlight our general observations below:

- *maxIter*: On the full dataset, maxIter 15 performs best. On downsampled data, maxIters of 20 or 25 perform slightly better with all other params held constant. This indicates slight overfitting at maxIters > 15 on the full dataset (see Table 3).
- *rank*: As expected, the higher the rank, the better the MAP. Therefore, the main tradeoff here is performance versus runtime. With the updated sparkSession configs, we are able to train and score rank 200 models in about 1.5 hours. While higher ranks result in higher MAPs, rank 200 still results in quite strong performance (see Table 4), so this seemed to be a good balance of the tradeoffs.
- *regParam*: regParam has less independent influence on model performance than some of the other hyperparameters except at the extremes of the spectrum (i.e. 100 was far too much regularization and resulted in quite poor performance regardless of other hyperparameters). However, at higher ranks (i.e. ≥ 200) , higher regParam values like 1 or 10 did result in slightly higher MAPs compared to smaller regParam values. This is likely due to the fact that at high ranks and low regParams, the model tends to overfit, so the additional regularization tempers this and boosts performance on validation data (see Table 5).
- *alpha*: After rank, alpha is the most independently influential hyperparameter. There is a clear parabolic relationship between alpha and MAP with MAP increasing with higher alphas until around 200. When alpha is greater than 200 performance drops. This trend is more pronounced at higher ranks likely because we benefit from higher weight on count data when the model has more flexibility in learning latent factors.

2.4 Data Transformation and Final Results

Given the right skew of play count data and the poor performance when downsampling based on popular items, we test different data transformations to limit model bias towards the most popular items. Using pyspark.sql.functions, we test square root, log, and log1p. As shown in Appendix C Table 7, square root and log1p both out-perform non-transformed data when using combinations of high rank and alpha. Given these were the most promising ranges for these hyperparameters, we did a significant portion of tuning using transformed data. Log without adding one performs poorly given all single play counts are treated as zeros.

The best hyperparameter combinations we found through tuning on 25% randomly downsampled data are maxIter = 15, regParam = [0.1, 1], rank = 200, and alpha between 75 and 150. Therefore, we train models in these hyperparameter ranges on both the square root and log1p transformed using full training datasets and apply these models to the validation data. Results of training using the close to 50 million training records are in Appendix D. The ultimate best combination is maxIter = 15, regParam = 1, rank = 200, and alpha = 150 on the square root data, which results in a MAP of 8.29% on the validation data.

We finally apply this model trained on the full training dataset and achieve a very similar MAP on the test data of 8.26%.

3 BASELINE MODELS

In order to baseline our recommendation model, we start with the simplest method: uniformly recommending the most popular songs. For this baseline we consider four strategies for measuring popularity: total play count, total interaction count, mean play count, and mean play count with user biases normalized. All of this was implemented in PySpark using fundamental data frame manipulation. The results show that the interaction count and play count popularity models give the highest MAP scores of 2.1% and 2.0% respectively. The results for the mean play count and user normalized mean play count are considerably lower at .00012% and .5% respectively.

Given the improvement to the mean ranking method from centering user biases, we hypothesize that a more general bias model with the right dampening parameters could out perform the interaction count popularity model. We also consider different transformations of the utility matrix. We employ a bias ranking algorithm in PySpark that determines user and item biases consecutively with a dampening parameter gridspace on a log10 scale from zero to one-hundred million. As shown in Appendix D Figure 4, increasing dampening parameters improves model performance with the highest user and item biases achieving a MAP of 1.89%, nearly as good as the popularity model. However, given these damping parameters are orders of magnitude above the average user interaction count, the model is effectively approaching a popularity model. Hence we conclude that the best way to baseline a recommendation system in this context is to simply rank on interaction count.

For a point of comparison, we also employ the lenskit bias model which implements a similar algorithm. To achieve this we build a custom singularity container on Greene and access the data in a scratch folder. A SLURM job is submitted capable of running the python code with the requested resources. The job returns the item rankings for each parameter set which is pipelined to code in PySpark capable of determining MAP scores, as lenskit has no built-in MAP evaluation. The results confirm the trends seen in the PySpark implementation, however the MAP scores are not as high as the PySpark implementation. We hypothesize this is because lenskit's algorithm learns the item bias parameter before optimizing the user bias, which is less ideal for creating user rankings. Appendix D includes visualizations of performance based on user and item bias dampening parameters for the PySpark and lenskit implementations.

4 COLDSTART

The goal of a cold start model in the context of song recommender is to learn latent factors of a trained ALS model using track metadata. With this infrastructure in place, our model has the flexibility to recommend songs not in the training data.

4.1 Generating Metadata Features

In order to generate metadata features for each track, we use the additional files from the Million Song Database, specifically the track_metadata.db and artist_term.db. Other files we have access to are missing data for significant portions of tracks and therefore do not suffice. Features from track_metadata.db are already assigned at the track level, but data in the artist_term database is only at the artist level. We generate features based on artist data by taking the most popular term associated with each artist and mapping it to all trackIDs of the associated artist. We note that we also tested mapping the top two terms per artist, but saw a decline in our final model performance. Each term is ultimately a binary feature used in our latent factor mapping. See Appendix E for a more detailed summary of features created from metadata.

4.2 Metadata to Latent Factor Mapping

After building the metadata feature table, we map each item from the known feature space into the latent feature space. To do this, we take the latent item factors from one of the top performing basic recommender models (parameters: rank=200, regparam=1, alpha=100, and maxIter=20). We then take all items in this V matrix, hold out 20% and then relearn these held out latent features using the other 80% of items. We use the item and user factors from a pre-trained ALS model for comparison purposes and so we are not retraining on only 80% of the data. We initially test Ridge and Lasso regression using the sklearn.multioutput module, which enables regression with multi-dimensional outputs. Both Ridge and Lasso require such high regularization that all new latent factors approach 0, so we also test kNN with k=1. Due to high computational costs that limit our ability to test, we do not test using other values of k. In practice for each held out item, this approach finds the most similar item in the training data and assigns the same latent factors to the held out item to create a new item matrix V'.

4.3 Updated Predictions and Evaluation

We generate new predictions using the same methodology discussed in the 'Basic Recommender System', where we calculate the similarity score between items and users and select the highest ranked items. We do this by using our updated item factor matrix V' and the same user factor matrix U. We do the similarity scoring implementation locally using NumPy's matrix multiplication, going user by user to avoid computational limitations. For the 10,000 validation users, this process takes approximately 10 minutes to run. Once we have generated updated predictions for each user, we evaluate these against the validation set using Spark.

4.4 Results

The initial model used here had a MAP of 7.2%, which we treat as the ceiling for our coldstart predictions. Given that we are relearning the latent features from this model, we do not reasonably expect to beat this performance. We estimate a floor for our coldstart predictions by calculating the MAP when using only the 80% of items from our training data and excluding all coldstart predictions, and get a score of 6.2%. After numerous iterations of re-adding the held out and re-learned items, we are unable to beat this floor and our highest MAP was 0.47% when using Lasso regression with a regularization parameter of 10000. We hypothesize this poor performance is due to our regression model poorly learning the latent factors with too much emphasis, thus these new similarity scores are arbitrarily high and the held out items are over-recommended. Based on these results, the optimal strategy would be to not recommend any coldstart items barring a vast improvement to this model. For future iterations we would look to improve our metadata features to more accurately convey the most important information about the coldstart items.

5 COMPARISON TO SINGLE-MACHINE IMPLEMENTATION (SMI)

After building the basic recommender model using Spark's ALS implementation, we attempt to employ lenskit's ALS tool on a single machine for a point of comparison. For a true single machine implementation, the data would need to be portable enough to store locally. Since this was not feasible, we intended to demonstrate the performance of this system on a resource constrained SLURM request. This would mimic a SMI while still having access to the data. The workflow was optimized locally on .5% downsampled train and test data. We were able to fit and predict in approximately 3 minutes. This code was packaged into a SLURM request with a custom built singularity container with varying values for model rank and iterations. No model was able to complete in less than 24 hours. In our most extreme case a Lenskit's ALS algorithm specified with a rank of 5 and an iteration limit of 2 was unable to complete in 24 hours despite being allocated 64GB of memory and 4 nodes. Code is submitted for review.

CONTRIBUTIONS

Applicable for actual application as well as contributions to the write up

- Sara Price: Complete model pipeline testing, data transformation, hyperparameter tuning and tracking, test set evaluation
- Pedro Galarza: Bias Extension, Single Machine Implementation Extension, Singularity Environment Construction
- Kieran Sim: Initial Pipeline Setup on Downsampled Data, Coldstart Extension
- Jack Epstein: Coldstart Extension, Downsample Testing, Initial Baseline Modeling

APPENDIX

A DOWNSAMPLING

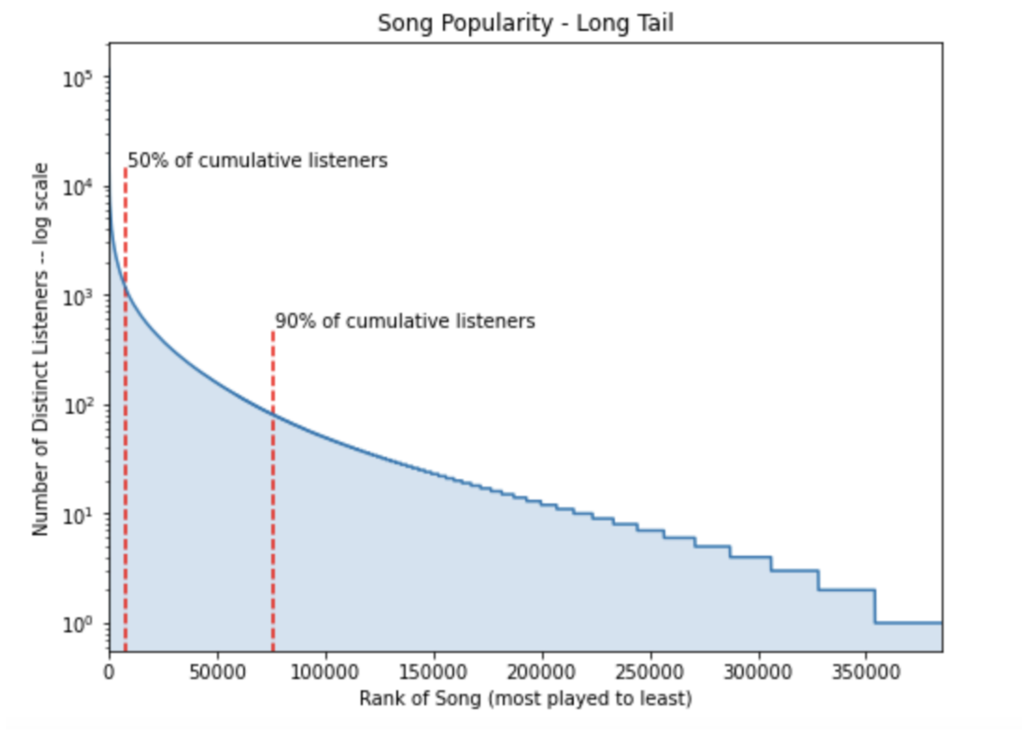

Fig. 1. Visualization of Right Skew in Song Popularity

Table 1. Comparison of Downsample Methods Using Rank = 100, Alpha = 5, MaxIter = 15, regParam = 0.1

Downsample Method	MAP	Approximate Runtime (hours)
All Validation Data + Random 10% Downsample of Non-Validation Data	3.94%	1.25
All Validation Data + Random 25% Downsample of Non-Validation Data	4.79%	1.5
All Validation Data + 10% Downsample by Popular Items	1.09%	1
All Validation Data + 25% Downsample by Popular Items	2.48%	1.25

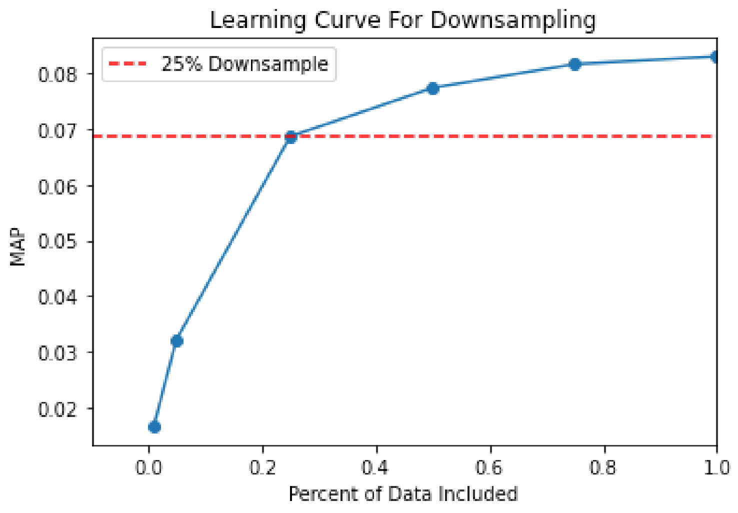

Fig. 2. Learning Curve for Randomly Downsampled Data

B HYPERPARAMETER TUNING

Table 2. Ranges of Hyperparameters Tested

Parameter	Values
maxIter	10, 15, 20, 25
regParam	$1e^{-05}$, $1e^{-03}$, 0.01, 1, 10, 100
rank	10, 25, 50, 100, 200, 300
alpha	0.5, 1, 5, 10, 25, 50, 75, 100, 150, 200, 300, 500

Table 3. MAPs for Different maxIter Values on 25% Random Downsampled Data

rank	regParam	alpha	MAP		
			maxIter 15	maxIter 20	maxIter 15
100	0.001	200	5.58%	5.99%	6.08%
100	0.1	200	5.61%	5.88	5.99%
100	1	200	5.61%	6.12%	6.22%

Table 4. MAPs and Runtime for Different Ranks on 25% Random Downsampled Data (maxIter = 25 and regParam = 0.1)

rank	alpha	MAP	runtime (in hours)
50	100	5.23%	0.53
100	100	6.12%	1.15
200	100	6.95%	2.17
300	100	7.25%	3.15

Note that runtimes are from spark Sessions run using increased parallelism noted in Section 2.3

Table 5. MAPs for Different regParam Values on 25% Random Downsampled Data

rank	alpha	regParam	MAP
100	100	0.00001	5.82%
100	100	0.001	5.82%
100	100	0.1	5.85%
100	100	1	6.08%
100	100	10	6.28%
100	100	100	2.83%

Table 6. MAPs for Different Alpha Values on 25% Random Dowsampled Data (maxIter = 20)

rank	regParam	alpha	MAP
100	1	1	3.92%
100	1	5	5.37%
100	1	10	5.53%
100	1	25	6.04%
100	1	50	6.24%
100	1	100	6.29%
100	1	200	6.12%
100	1	500	5.52%

C DATA TRANSFORMATION

Table 7. Comparison of Transform Methods across a range of rank and alpha values on 25% Random Downsampled Data (maxIter = 20, regParam = 1)

rank	alpha	MAP		
		No Transform	Log1p	Square Root
100	25	6.04%	5.92%	6.13%
100	50	6.24%	6.37%	6.51%
100	100	6.29%	6.64%	6.69%
100	200	6.12%	6.66%	6.60%
200	25	6.75%	6.54%	6.78%
200	50	7.10%	7.02%	7.22%
200	100	7.24%	7.35%	7.43%
200	200	7.19%	7.39%	7.34%

Table 8. Validation Results Using Full Training Data

maxIter	rank	regParam	alpha	MAP	
				Square Root	Log1p
15	200	0.1	75	8.13%	7.99%
15	200	0.1	100	8.24%	8.11%
15	200	0.1	150	8.30%	8.25%
15	200	1	75	8.07%	7.99%
15	200	1	100	8.13%	8.06%
15	200	1	150	8.18%	8.15%
20	200	0.1	75	8.10%	7.98%
20	200	0.1	100	8.19%	8.20%
20	200	0.1	150	8.25%	8.20%
20	200	1	75	8.01%	7.94%
20	200	1	100	8.06%	8.00%
20	200	1	150	8.10%	8.08%

D BIAS MODEL

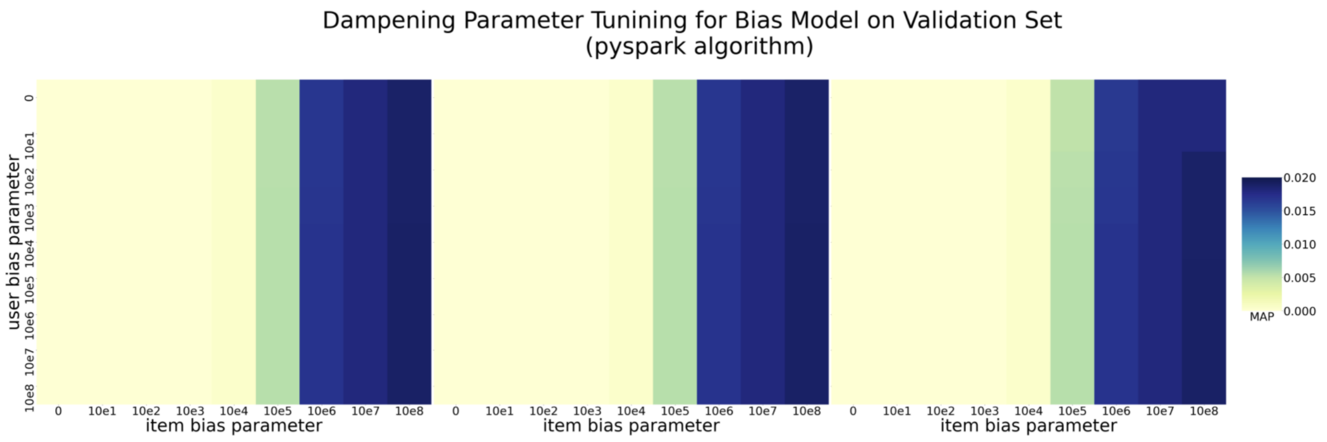

Fig. 3. Parameter Tuning Bias Model on Validation Set (PySpark)
a. log transformed, b. Square Root Transformed, c. Raw Data

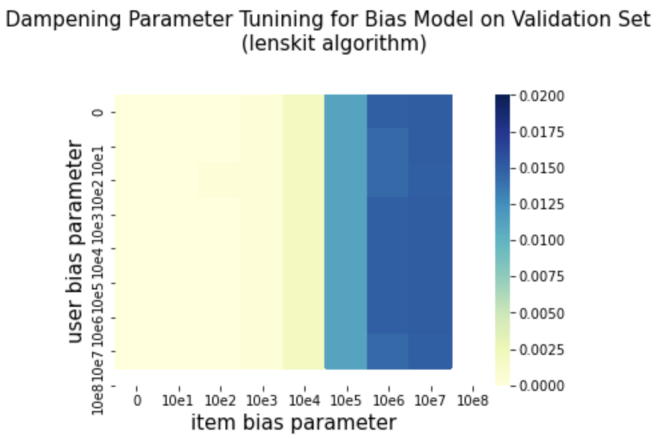

Fig. 4. Parameter Tuning Bias Model on Validation Set (LensKit)

E COLD START

Table 9. Meta Features Used for Coldstart

Feature	Description
Duration	Total time in seconds of the track
Artist_Familiarity	float in [0,1] estimating artists level of fame
Artist_Hotness	float in [0,1] estimating artists level of popularity
Year	Year of the track's release
Artist_Terms	359 binary features indicating if the track's artist is associated with the term. Terms include "Rock", "Hip Hop", etc.

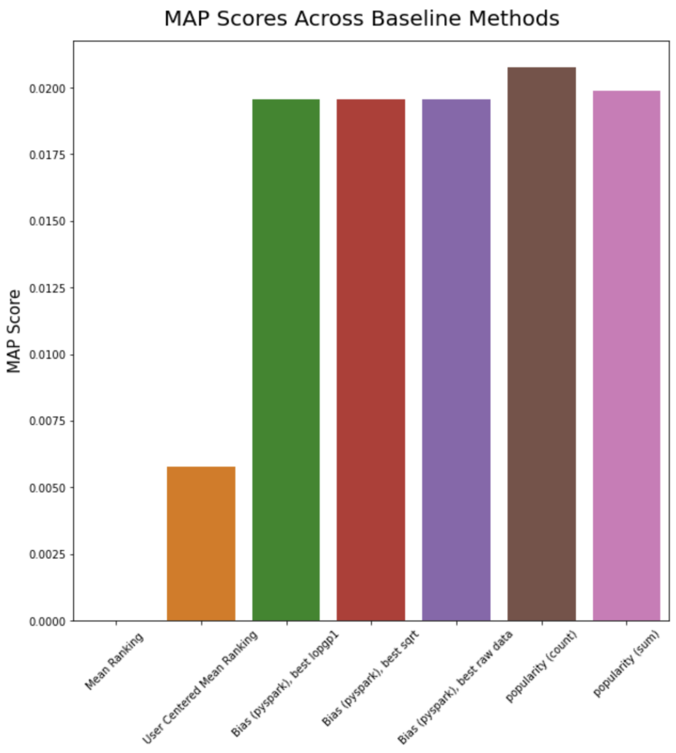

Fig. 5. Test Set Evaluation Metrics for Baseline Methods

Citations

(1) Y. Hu, Y. Koren and C. Volinsky, "Collaborative Filtering for Implicit Feedback Datasets," 2008 Eighth IEEE International Conference on Data Mining, 2008, pp. 263-272, doi: 10.1109/ICDM.2008.22.