Recommendation System for Music

DS-GA 1004: Big Data Spring 2023 – Group 22 – Github

Hoa Duong
Center for Data Science, NYU
hhd2020@nyu.edu

Ridhika Agrawal
Center for Data Science, NYU
rra10001@nyu.edu

ABSTRACT

This project explores users' listening history to build a collaborative-filtering song recommendation system. First, a popularity model is built as a baseline using two methods of determining popularity. Then, a Latent Factor Model is built using the Alternating Least Square (ALS) algorithm provided in Spark's library. Additionally, a single-machine approach using LightFM is implemented for comparison.

1 Dataset

1.1 Data Description

We use the ListenBrainz datasets, which contain implicit feedback from music-listening behaviors. The datasets are pre-partitioned into training and testing datasets: the training data were collected between 2015 and 2018, and the testing data were collected in 2019. The ListenBrainz datasets consist of 'tracks', 'interactions', and 'users' datasets. For purposes of this project, only the 'tracks' and 'interactions' data are utilized. Within this report, tracks and songs are used interchangeably, and so are listeners and users.

1.2 Data Pre-processing

The 'tracks' dataset contains the 'recording_msid', 'track_name', 'artist_name', and 'recording_mbid', which are all in strings format, where each row denotes a unique song. The 'interactions' dataset contains 'user_id', which is a unique integer, 'recording_msid', and 'timestamp', where each row represents each time the user listens to that song. As we require numerical IDs for the ALS algorithm, we use SQL Window function to assign 'song_id' to the unique tracks in this dataset, and use this to identify the user-song interactions instead of 'recording_msid'.

Within the 'interactions' dataset, there were no null values in the 'user_id' and 'recording_msid' or 'song_id' columns. We decided to drop duplicated rows (duplicated in terms of 'user_id', 'song_id', and 'timestamp'.) The initial dataframe consists of 179,466,123 rows. After dropping duplicates, the final dataframe has 179,451,196 rows, which is what we use for partitioning.

The same 'song_id' generated by the SQL Window using the training dataset is used to identify the user-song interactions in the testing dataset as well. It is worth noting that some tracks might only appear in the testing dataset and not the training dataset, and thus would not have an associated 'song_id', and would be considered a cold-start problem. Within the scope of this project, we do not consider cold-start songs.

1.3 Data Partitioning

To ensure we can properly tune hyperparameters, we partition the given training data into training and validation sets. Each row in the dataset consists of a user-song interaction. We first group the data by each unique user. For users with only one entry, we assign that user to the training dataset to avoid cold-start. For users with more than one entry, we do a random 80-20 training-validation split. After partitioning, there are 143,564,094 interactions in the training dataset, consisting of interactions from 7,909 unique users and 25,630,475 unique songs. On the other hand, there are 35,887,102 interactions in the validation dataset, consisting of interactions from 7,761 unique users and 11,653,886 unique songs.

1.4 Data Filtering

We first compute a compact utility matrix, counting the number of interactions between each pair of user-song as implicit feedback, creating a dataset containing 'user_id', 'song_id', and 'total_count', where 'total_count' represents the number of times the user listens to this song. By using the count of interactions for each user, instead of a simple binary indicator of interaction, we can capture when users listen to a song multiple times, to better measure popularity. The resulting compact utility matrix for training has 55,190,069 rows and for validation has 21,318,837 rows.
To aid with training the models, hyper-tuning the parameters, and comparing model efficiency in later stages, we create several versions of filtered training datasets. Specifically, we keep the songs with more than (i) 3 unique listeners (> 3), (ii) 10 unique listeners (> 10), (iii) 15 unique listeners (> 15), and (iv) 30 unique listeners (> 30). There are 25,630,475 songs in the training dataset, of which (i) 2,306,965 songs have > 3 listeners, (ii) 621,736 songs have > 10 listeners, (iii) 376,796 songs have > 15 listeners, and (iv) 145,034 songs have > 30 listeners. After filtering, the size of the utility matrix for training reduces from 55,190,069 rows to (i) 26,660,473 rows, (ii) 17,008,753 rows, (iii) 13,906,998 rows, and (iv) 9,000,305 rows.

2 Popularity-Based Model

2.1 Popularity-Based: Average Utility

From the utility matrix, we calculate the popularity score for each song as follows:

$$P[i] = \frac{\sum_u R[u, i]}{|R[:, i]| + \beta},$$

where $P[i]$ denotes the popularity score for song i and $R[u, i]$ denotes the number of interactions between user u and song i, i.e., the number of times user u listens to song i. We then take the top 100 songs, ordered by popularity score as the recommendations.

By using the average number of listens, we avoid over-rating a song that is listened to many times but by just one or two users, as being popular. Additionally, with the damping factor β, we can control how much we want to penalize this instance, as the effect of β will be amplified for a lower number of users who interact with a song.

However, by dividing the total number of interactions of a song by the number of users who interact with that song, we are implicitly penalizing the song for having many users listening to it, if all users listen to it equally many times.

2.2 Popularity-Based: Weighted Utility

To account for the implicit penalty of the model using average utility, we explore a model using a weighted utility. Particularly, we multiply the total number of interactions of a song by the number of users who interact with that song:

$$P[i] = \sum_u R[u, i] \times |R[:, i]|$$

2.3 Model Evaluation

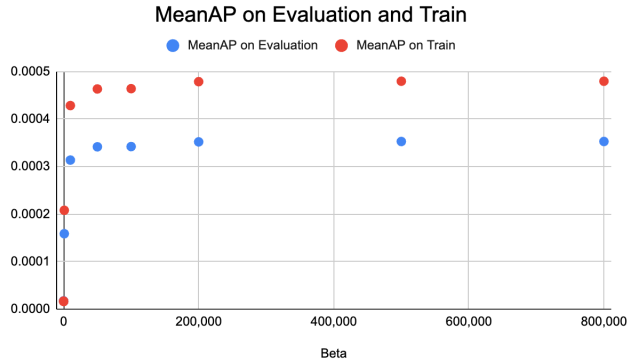
We use Mean Average Precision (meanAP) and Normalized Discounted Cumulative Gain (NDCG) at 100 to evaluate the performance of both models. Precision is calculated as the number of + interactions divided by the number of total interactions. Average precision is the weighted mean of precision at each additional data point. meanAP at 100 is the average precision scores for the top 100 results. meanAP ranges from 0-1, with 1 being the best score. NDCG is calculated by dividing the discounted cumulative gain of a ranked list by the discounted cumulative gain of the "ideal" list of items. NDCG also ranges from 0-1, with 1 being the best score.

2.3.1 Popularity-Based: Average Utility

The meanAP on the validation set is 0.00035 with a damping factor (β) of 500,000. The NDCG on the validation set is 0.0039 with a β of 500,000.

2.3.1.a Hyperparameter Tuning

We train on the training set using β from 10 to 800,000 and tune the hyperparameter using the validation set. The meanAP on the validation set increases at a decreasing rate as β increases until a beta of 500,000, and then decreases slowly towards 800,000, as can be seen below. Based on the results, we decided to set β = 500,000.



We train the data to maximize for meanAP at 100 because it assumes that users only care about the top 100 results. In this music recommendation system, we believe that it is more valuable to get the top 100 songs to be as correct as possible, as opposed to having an overall list. meanAP accounts for this by penalizing for having an incorrect song in the top 100 recommended songs.

2.3.2 Popularity-Based: Weighted Utility

The meanAP on the validation set is 0.00049. The NDCG on the validation set is 0.0045.

3 Latent Factor Model: Spark

3.1 ALS Implementation

To build the Latent Factor Model, we used Spark's alternating least squares (ALS) algorithm. The ALS algorithm uses matrix factorization to find latent features between users and items in order to make personalized recommendations. It does so by first converting the utility matrix into two simpler matrices that represent the latent factors for users and items. Next, it fixes one of the matrices and solves for the other to minimize the error based on actual vs. predicted rating. It then fixes the other matrix, and calculates the other one. The algorithm alternates between solving for the two matrices. ALS is a popular algorithm for song recommendations, and is built

to handle large, sparse datasets in an efficient manner using parallel computing.

3.2 Hyperparameter Tuning

To tune the ALS hyperparameters, we use the dataset with only songs which have been listened to by more than 3 unique users. Specifically, we run the models on the following combinations of hyperparameters:

Table 1: Hyperparameters for > 3 users training dataset

Parameter	**Values**
rank (dimensionality)	[10, 20, 30 40]
regParam (regularization)	[0.001, 0.1, 0.15, 0.2, 0.5, 1]
alpha (implicit feedback)	[0.1, 0.5, 0.75, 1.0, 1.5, 2.0, 5.0]

We start with a wide range of values and narrow down to the specific values which provide the best improvement. From Appendix Table 1, we see that generally, with the same rank and alpha values, meanAP increases along with regParam until around 0.1. On the other hand, with the same rank and regParam values, meanAP increases along with alpha until around 1.5. The best results we get are with increasing the rank, but the size of the data did not allow us to run with ranks higher than 40 for this version of the data.

Thus, in order to get a better model, we train on the training set after retaining only songs with > 10 unique users. We implement the models with the following hyperparameters:

Table 2: Hyperparameters for > 10 users training dataset

Parameter	**Values**
rank (dimensionality)	[50, 60, 70, 80, 90, 100, 120, 140, 150, 160, 200, 250]
regParam (regularization)	[0.15, 0.2, 0.25, 0.5, 1.0]
alpha (implicit feedback)	[1.0, 1.5, 2.0]

From Appendix Table 2, we see that with the same rank and regParam, meanAP increases as alpha increases from 1.0 to 1.5, but decreases as alpha further increases to 2.0. Thus, we conclude that alpha = 1.5 is the best value. Similarly, with the same rank and alpha, meanAP increases as regParam increases up to 0.5, then decreases as regParam increases to 1.0. Thus, we conclude that regParam = 0.5 is the most appropriate value. Increasing rank increases meanAP as well, however, given the size of the data and the limited resources in terms of time and cluster allocation, we could not test higher ranks.

We get a meanAP at 100 of 0.05678 when rank = 250, regParam = 0.5, and alpha = 1.5 on the validation set. The NDCG for this parameter combination is 0.15088.

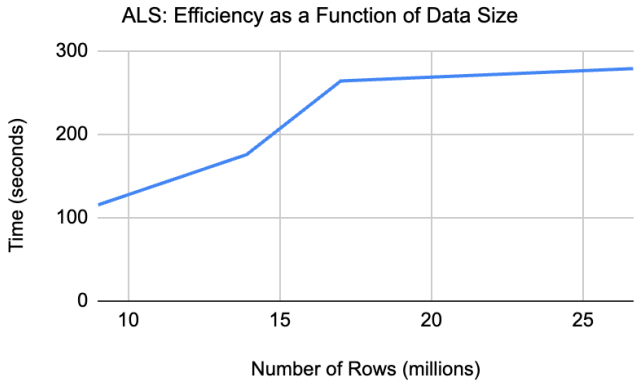
3.3 Evaluation on Test Data

The popularity-based model using average utility with β = 500,000 evaluated on the testing set gives meanAP of 0.000082 and NDCG of 0.0013. The popularity-based model using weighted utility gives meanAP of 0.000099 and NDCG of 0.0011.

The meanAP at 100 on the test set with the tuned latent factor model (rank = 250, regParam = 0.5, and alpha = 1.5) is 0.010284 and the NDCG is 0.043413. Compared to the baseline, we can see that personalizing recommendations based on user interactions increases the model performance significantly, with meanAP increasing by 125 times compared to the average-utility method and 103 times compared to the weighted-utility method, and NDCG by 33 times compared to the average-utility method and 40 times compared to the weighted-utility method.

3.4 Efficiency

To evaluate the efficiency of the ALS algorithm, we get run times from training on data filtered with each song having >3, >10, >15 and >30 unique users. We train the model with the following parameters: rank = 40, alpha = 1.0, RegParam = 0.2 and maxiter = 10. Although this is not the most optimal set of parameters, we use this to get an accurate time reading, which would otherwise have not worked for the bigger datasets. The results show that as the size of the data increases, the time increases at a decreasing rate. This happens because as the dataset increases, the algorithm is able to make full use of parallelizing tasks. Adding additional data does not make much difference to the run time.



4 Latent Factor Model: Single Machine

4.1 LightFM Implementation

We run the latent factor model using another recommendation algorithm, LightFM. This also is a collaborative-filter-based recommender system, which we run on our local machine. It works by creating a low-rank approximation of the interactions data. The algorithm trains a neural network to predict the probability of a user interacting with a song, given the user's preferences as well as the item features. It minimizes a hybrid loss function, which is a combination of loss for implicit and explicit feedback. It also works well on large, sparse data and is scalable. To that end, LightFM works well for our use case since in the songs recommendation system, we rely on implicit feedback and have a large dataset. We compare LightFM with Spark's ALS algorithm on evaluation metrics and on efficiency as a function of data size.

4.2 Evaluation on Test Data

The final LightFM model has the following parameters: loss = warp and epochs = 50. To evaluate the model on the test data, we used precision_at_k where k = 100. After training on the filtered data set (> 3 unique users per song and > 10 songs respectively), we got 0.11091 and 0.13599 as precision at 100 and on test data.

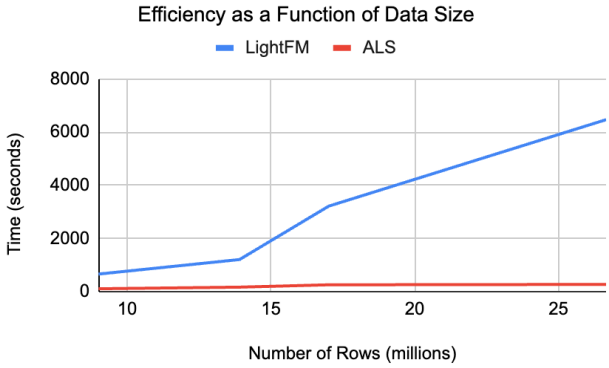
4.3 Efficiency

To understand the system's efficiency, we trained LightFM on the utility matrix, consisting only of songs that have been listened to by >3, >10, >15 and >30 unique users. We find that as the number of rows in the dataset that the model is being trained on increases, the time it takes to train the model (on num_threads = 1) increases at an increasing rate. The increase is almost exponential. This makes sense since there can be no parallelization.

4.4 Comparing with Spark

The ALS precision at 100 on the test set is 0.037. Compared to ALS, LightFM precision is higher by 0.1. The direction of difference is not too unexpected, since the ALS model was not trained to its full capacity because of computational constraints. However, the magnitude of the difference is surprising.

As can be seen below, the results from LightFM models take longer to run than ALS. This difference comes from the fact that Spark's distributed calculations across different machines. As a result of the behavior of a single machine being overburdened by increasing data, and multiple machines being able to handle additional data better, the ratio of the difference between LightFM and ALS increases at an increasing rate.



5 Conclusion and Future Work

Overall, the latent factor model with the ALS algorithm gives a mean Average Precision at 100 of 0.0103, with a performance improvement of over 100 times compared to the popularity-baseline model. This emphasizes the immense effectiveness of personalized recommendation systems, which takes advantage of specific user-song interactions and considers users' preferences. Additionally, compared to the Spark's ALS algorithm, the single-machine implementation of latent factor model with LightFM takes significantly longer to train and evaluate, due to its inability to parallelize. However, tuning hyperparameters for Spark's latent factor model with the ALS algorithm also requires lots of resources.

Outside the scope of this report, we believe that the model can be improved in several ways. First, we can improve the hyperparameter tuning by running models with an increased rank, provided we have more computational power and resources. Currently, we are filtering by the number of unique listeners of a song. We can further pre-process the data to filter down the users who have not interacted with more than 3 unique songs to improve the recommendation system. Additionally, our current model does not handle cold-start users or songs. Thus, an improved model should explore how to handle cold-start users or songs, such as utilizing background popularity models, gathering more data about users' demographics (gender, age group, etc.) from ListenBrainz or external sources to make use of users' similarities, or making use of tracks metadata like artist's name or albums.

NOTES

[1] We worked on the entire project side-by-side.

[2] We only report two implementations of the baseline popularity-based model for simplicity.

[3] To keep the report succinct, we only report some combinations of hyperparameters in each case. A more detailed summary can be found on our Github page.

APPENDIX

Table 1: Hyperparameter Tuning Results for Data with Each Song Having at least 3 Unique Users

Rank	RegParam	Alpha	MeanAP	NDCG
10	0.001	0.1	0.00237	N/A
10	1.0	0.1	0.00204	N/A
10	0.001	1.0	0.00342	N/A
10	0.1	1.0	0.00377	N/A
10	0.0001	1.5	0.00351	N/A
10	0.1	5	0.00312	N/A
20	0.2	1.0	0.00723	0.03525
20	0.2	2.0	0.00685	0.03504
30	0.1	1.0	0.00857	0.04046
40	0.1	1.0	0.01036	0.04602
40	0.1	1.5	0.01039	0.04671
40	0.15	1.0	0.01120	0.04798
40	0.15	1.5	0.01108	0.04846
40	0.2	0.15	0.01073	0.04087
40	**0.2**	**1.0**	**0.01198**	**0.04974**
40	0.2	1.5	0.01176	0.04087
40	0.2	2.0	0.01161	0.05020

Table 2: Hyperparameter Tuning Results for Data with Each Song Having at least 10 Unique Users

Rank	RegParam	Alpha	MeanAP	NDCG
50	0.2	1.0	0.01374	0.05746
60	0.2	1.0	0.01586	0.06344
70	0.2	1.0	0.01769	0.06873
80	0.2	1.0	0.01968	0.07397
90	0.2	1.0	0.02154	0.07872
100	0.2	1.0	0.02343	0.08326
120	0.15	1.0	0.02405	0.08644
120	0.2	1.0	0.02677	0.09117
120	0.25	1.0	0.02849	0.09524
140	0.2	1.0	0.02985	0.09835
150	0.15	1.0	0.02790	0.09579
150	0.25	1.0	0.03341	0.10615
160	0.2	1.0	0.03268	0.10471
200	0.15	1.0	0.03363	0.10906
200	0.2	1.0	0.03769	0.11577
200	0.25	1.0	0.04081	0.12165
200	0.25	2.0	0.03729	0.11723
250	0.2	1.0	0.04327	0.12823
250	0.25	1.5	0.04493	0.13247
250	0.25	2.0	0.04277	0.12924
250	0.5	1.0	0.05584	0.14494
250	**0.5**	**1.5**	**0.05678**	**0.15088**
250	0.5	2.0	0.05509	0.15035
250	1.0	1.5	0.05359	0.13880