DSGA 1004 Final Project - Recommender System

Fiona Chow, Xinyue Ma, Christine Gao

GitHub: https://github.com/nyu-big-data/final-project-group16

1 DATA PRE-PROCESSING

The steps we took to pre-process the full data include:

(1) Missing value imputation - replacing missing 'recording_mbid' with the associated 'recording_msid'.
(2) Omitting metadata or irrelevant columns
(3) Keeping the top 3% of tracks that have the most unique listeners
(4) Dropping users where number of unique songs listened to is less than 10

One choice that we made to improve the efficiency and stability of our recommendation system was to drop tracks with few unique listeners and users who interact with few unique tracks. To set the thresholds, we looked into the distribution of those quantities demonstrated in Table 1 and Table 2. We removed approximately 97% of tracks with less than 10 unique listeners since they had limited engagement. Additionally, we filtered out around 3% of users who had interacted with fewer than 10 songs to ensure an adequate level of user activity. These filtering steps helped streamline the dataset for subsequent analysis and modeling.

# Unique Listeners Per Track	Track Counts
[0, 5)	21,869,058
[5, 10)	1,137,840
[10, 20)	512,031
[20, 30)	156,828
[30, 40)	73,404
[40, 50)	41,066
≥50	100,969

Table 1: Distribution of number of unique listeners per track

# Unique Tracks Interacted Per User	User Counts
[0, 10)	236
[10, 25)	111
[25, 50)	156
[50, 100)	158
[100, 500)	981
≥500	6,550

Table 2: Distribution of number of unique tracks interacted per user

2 TRAIN/VALIDATION PARTITIONING

To support evaluating the average performance across users, we randomly partitioned the interactions of each user into train and validation datasets. To ensure each user has at least one record in the training set and the validation set, we grouped the dataset by 'user_id'. Within the subgroups of 'user_id', each record was assigned its row number and partitioned into either the training set or validation set based on the total number of records each subgroup had. For example, if user_1 interacted with 5 songs, those records were assigned row numbers from 1 to 5; the records with row numbers less than 4 (5*0.8) were assigned to the training set, and the rest records were assigned to the validation set. This method ensured a random partition and an approximately 80:20 split between the training dataset and the validation dataset.

To address the cold-start user problem in our study and reserve the test set for final evaluation, we simulated the presence of cold-start users in the validation set. We accomplished this by selecting a subset of users who ranked in the bottom 10 percentile of total interactions and excluding all their interactions from the training data. This simulation allowed us to assess the performance of the cold-start model on the validation set while ensuring the test set remained untouched for comprehensive evaluation.

Once the users for the cold-start simulation were identified, we proceeded to divide our train and validation sets into two groups: users facing the cold-start problem and users not facing the cold-start problem. We followed a similar partitioning approach for our test set, with the distinction that we did not require any simulation for cold-start users. Instead, we identified cold-start users as those present in the test set but absent from the train set. This partitioning allowed us to evaluate the performance of the cold-start model separately for both user categories while maintaining the integrity of the test set.

Our popularity baseline model and latent factor model were developed using the users who did not encounter the cold-start problem. In contrast, the cold-start model specifically targeted and incorporated users who faced the cold-start challenge. This segregation allowed us to tailor the models accordingly and address the unique characteristics of each user group.

3 POPULARITY BASELINE MODELS

The formula we used for our popularity baseline model is given by:

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

where P[i] is the popularity score for a specific track (i), R is a matrix of implicit ratings between users (u) and tracks (i) , and β is a damping constant to prevent extreme recommendation scores for items with very few distinct user ratings. In addition, we conditioned our model on there existing an interaction between user and track.

We will hyperparameter tune β.

4 BASELINE MODEL PERFORMANCE

To evaluate our model's performance, we opted for Mean Average Precision at K (MAP@K). MAP@K calculates the average precision

at K for each user and then computes the mean across all users. This metric quantifies the fraction of relevant items present in the top K recommended items across the entire dataset. With K set to 100, we focused on evaluating the relevance of the top 100 songs recommended to each user. MAP@K is a widely recognized and reliable metric commonly used for assessing recommender systems. [4]

We hyperparameter tuned β from our popularity baseline model and the results of tuning the damping factor β are as follows in Table 3:

β	Non-Cold-Start	Cold-Start
1	6.74E-06	9.99E-06
100	3.31E-05	1.23E-05
1,000	2.47E-04	1.59E-04
5,000	3.467E-04	2.17E-04
10,000	3.58E-04	2.17E-04
20,000	3.67E-04	2.20E-04
100,000	3.76E-04	2.42E-04
1,000,000	3.84E-04	2.46E-04

Table 3: MAP@100 Results of Beta hyperparameter tuning on validation set

	Non-Cold-Start	Cold-Start
Val	3.467E-04	2.17E-04
Test	1.192E-04	4.618E-04

Table 4: MAP@100 Results of Baseline Popularity Model based on $\beta = 5,000$

The relatively small values of MAP@100 in our recommendation system can be attributed to the challenge of suggesting 100 previously unheard songs that users may like. This task is inherently difficult as it requires accurate predictions for unfamiliar preferences. Additionally, we believe that our recommendation system might be better suited for an alternative evaluation metric such as Normalized Discounted Cumulative Gain (NDCG). Unlike MAP, NDCG places greater emphasis on the quality of the top-ranked tracks, aligning with how users typically engage with ranked lists. NDCG is less affected by the distribution of relevant items throughout the list. Given more time, we would prioritize optimizing our system for NDCG to better reflect user preferences and interactions.

After analyzing the results presented in Table 3, we determined that β = 5,000 was the optimal value for the smoothing constant. This selection was based on a high evaluation score, as further increasing the value of β only resulted in marginal improvements for both non-cold start and cold-start MAP@100. Although β = 1,000,000 achieved the highest validation score, we opted against using it due to the dataset containing 7,909 users, making β = 1,000,000 appear excessively large as a smoothing constant. We aimed to avoid introducing significant bias into the model by acknowledging that some songs naturally enjoy higher popularity.

From Table 4, while the differences were not too big, baseline popularity model fared worse on the test data than on the validation data for the non-cold-start users. This observation and the relatively low scores motivates the exploration of alternative models that can enhance performance and exhibit improved generalizability.

5 LATENT FACTOR MODEL IMPLEMENTATION AND HYPERPARAMETER TUNING

We implemented the Latent Factor Model by leveraging Spark's alternating least squares (ALS) method. The latent factor model is used in recommendation systems to capture underlying user-item relationships by representing them in a latent feature space. We first fit the model using the training dataset. After fitting the model, we used the function recommendForUserSubset() to make the top 100 recommendations. The recommendations were compared with the ground truth using MAP@100.

Prior to hyperparameter tuning, we tried varying the 'maxIter' parameter. 'maxIter' refers to the maximum number of iterations that the algorithm will execute. If the algorithm reaches convergence before this number of iterations, it will stop, otherwise, it will stop after 'maxIter' iterations, regardless of if convergence has been achieved. It is important that a model converges as it means the algorithm has found a solution where the model's parameters are optimal, given the constraints of the algorithm and the data. Based on the documentation from the Apache Spark website [2], ALS typically converges within 20 iterations and given that 20 iterations was taking a long time to run on the cluster and not giving a better MAP@100 score, we chose 15 as the value for 'maxIter' to ensure the model converges. Interestingly, we observed a slight decrease in MAP@100 as we increased 'maxIter' from 10 to 12, and from 12 to 15.

To tune the hyperparameters, we ran through multiple iterations varying the 'rank', 'regParam', and 'alpha'. 'rank' is the number of latent factors in the model. Latent factors are underlying dimensions that explain observed variables. In a recommendation system, these can be considered as the underlying tastes and preferences of the users. 'regParam' specifies the regularization parameter in ALS which is used to control the magnitude of user and item factors, effectively shrinking their values and making the model less likely to fit the noise in the training data and more likely to generalize to unseen data. A higher 'regParam' indicates more regularization. 'alpha' is the degree of confidence in implicit preference observations with a higher 'alpha' indicating more confidence.

To streamline our parameter tuning process and work within the limitations of our available resources, we adopted a one-parameter-at-a-time approach instead of an exhaustive grid search involving all possible combinations. We prioritized investigating the rank parameter, as it determines the number of latent factors in the model and is deemed more critical than the other two parameters [1]. By conducting a more comprehensive exploration of the rank value, we aimed to identify the optimal setting for this parameter. Furthermore, given the limited resources of our cluster, conducting a grid search with multiple values for all three parameters might not have been feasible.

We conducted experiments using regParam=1 and alpha=1 while varying the rank parameter with values of 10, 50, 100, 150, and 200. Our observations revealed that increasing the rank led to higher MAP@100 scores, indicating improved performance. Based on these findings, we narrowed our focus to rank values of 100, 150, and 200 and proceeded to tune the regParam. Table 5 displays the rank and regParam values we experimented with, along with the corresponding MAP@100 results. Among the tested combinations, the model with rank=200 and regParam=0.5 achieved the best MAP@100 score of 0.02. As a result, we made the decision to set the rank parameter at 200 and the regParam parameter at 0.5 If time permitted, we would have further increased the rank to explore potential performance gains. However, due to longer execution times at rank=200, we concluded our hyperparameter tuning for rank at this value.

rank/regParam	0.1	0.5	0.8	1
100	0.0118	0.0135	0.0129	0.0119
150	0.0143	0.0171	0.0161	0.0146
200	0.0165	0.0200	0.0184	0.0165

Table 5: Tuning for rank and regParam (alpha=1)

Table 6 demonstrates the values of alpha we tried and the resulting MAP scores (while keeping the best rank and regparam we had just found constant) . From the results in Table 6 , we observe that MAP@100 achieves the best value with alpha 1.

rank	regParam	alpha	MAP
200	0.5	0.1	0.0054
200	0.5	0.5	0.0166
200	0.5	1	0.0200
200	0.5	10	0.0172

Table 6: Tuning for alpha (rank=200, regParam=0.5)

6 LATENT FACTOR MODEL PERFORMANCE

From the previous section, we achieved optimal performance with a rank of 200, regularization parameter of 0.5, and an alpha of 1. These parameters, identified through rigorous tuning, were then used to evaluate our model on unseen data - the test set. Key performance metrics, including precision and training time for both the validation and test sets, are detailed in Table 7. This analysis offers insights into the model's predictive accuracy and computational efficiency.

Metric	Val	Test
MAP@100	0.0200	0.0171
Training time	1190.00 s	581.66 s

Table 7: Latent Factor Model performance

We made some optimizations to our configuration between our validation and test run out of necessity due to the volatility of the cluster which resulted in improvements in the test time of the latent factor model as seen from Table 7.

In summary, the validation and test performance of our latent factor model are very similar, indicating that our latent factor model generalizes well to unseen data. The latent factor model also outperforms the popularity baseline model on both validation and test data by a lot as seen in Table 4, further validating that it is a more effective model in predicting top 100 songs to users.

7 EXTENSION: SINGLE MACHINE IMPLEMENTATION

We implemented LightFM as the single machine implementation on the dataset. LightFM is a Python implementation of hybrid matrix factorization, using collaborative filtering and content-based filtering techniques. The model learns latent interactions for users and items and can incorporate metadata into the matrix factorization algorithm. [3]

The 'Dataset' tool builds the interaction and feature matrices, creating mappings between user and song ids. We fit the Dataset a combined set of user and song ids from both the train and test sets, as training requires all users and songs to belong to both datasets.

This user-song interaction dataframe was then converted into a COO-matrix, a sparse matrix in the COOrdinate format, using the built in 'build_interactions' method from the lightfm package. The model was trained on the training split with a WARP loss function on a single core.

To compare the single-machine implementation, we evaluated MAP and efficiency, or number of seconds elapsed recorded by the CPU:

Metric	Val Small	Val Big
MAP@100	0.0289	0.0006
Training time	6.41 s	221 s

Table 8: LightFM performance

In comparison to ALS:

Metric	Val Small	Val Big
MAP@100	0.0108	0.0200
Training time	564.82 s	1190.00 s

Table 9: ALS performance

In terms of efficiency, we noticed that the training time for the lightFM model was significantly faster compared to the ALS model, regardless of dataset size. Additionally, when evaluating the performance using MAP@100, lightFM achieved higher scores on the small dataset, but lower scores on the large validation sets compared to the ALS model.

8 EXTENSION: COLD-START USERS IMPLEMENTATION

The second extension we chose to implement was the user cold-start problem. A user cold-start problem refers to the subset of users with no interaction history that enters the system. In other words, if there is a user in the test set that is not already in the training set, that user is a cold-start user.

One decision we made was to evaluate the user cold-start problem as a separate group from the other users. This disaggregation allowed us to see how our cold-start problem fared on its own. This required us to split our train and validation set to accommodate this decision. We have documented this in detail in the Train/Validation section.

To handle the cold-start users lacking demographic data in the dataset, we utilized the baseline popularity model from the non-cold-start users. Subsequently, we assessed the performance of these users by evaluating their top 100 songs in their respective validation and test sets.

The results in Table 3 reveal that cold-start users generally exhibit lower performance compared to non-cold-start users in the validation set. This outcome is unsurprising as we possess a larger amount of training data for the non-cold-start users, while the cold-start users lack any available information.

In contrast to the results observed in the validation set, the test set reveals that cold-start users outperformed non-cold-start users, as indicated in Table 4. Moreover, the cold-start model exhibited improved performance in the test set compared to the validation set. Collectively, these findings suggest that the cold-start model exhibits a certain degree of performance volatility when applied to unseen data. In practical applications, addressing the volatility in the cold-start model's performance could be achieved through the strategic utilization of user demographics and active learning techniques.

9 CONTRIBUTIONS

Xinyue Ma: Preprocessing, training and validation split, Latent Factor Model implementation, ALS hyperparameter tuning
Fiona Chow: Popularity Baseline Model, Evaluation, User Cold-Start extension, ALS hyperparameter tuning
Christine Gao: Preprocessing, LightFM model extension, ALS hyperparameter tuning

REFERENCES

[1] Gaurav Dhama. *Tuning Hyper Parameters ALS Model.* URL: https://stackoverflow.com/questions/48779687/tuning-hyper-parameters-als-model. accessed May 13, 2023.

[2] The Apache Software Foundation. *Collaborative Filtering - RDD-based API.* https://www.apache.org https://spark.apache.org. URL: https://spark.apache.org/docs/2.2.0/mllib-collaborative-filtering.html. accessed May 13, 2023.

[3] Maciej Kula. *Welcome to LightFM's documentation!* URL: https://making.lyst.com/lightfm/docs/index.html#. accessed May 13, 2023.

[4] ML Nerds. *MAP at K : An evaluation metric for Ranking.* URL: https://machinelearninginterview.com/topics/machine-learning/mapatk_evaluation_metric_for_ranking/. accessed May 15, 2023.