Recommendation System Final Project Report

Jialing Li
Center for Data Science
New York University
New York, NY, USA
jl9716@nyu.edu

Kristine Zeng
Center for Data Science
New York University
New York, NY, USA
yz4792@nyu.edu

1 INTRODUCTION

Recommendation systems play a crucial role in providing personalized suggestions for various online activities. Among the effective recommendation approaches, collaborative filtering stands out. Collaborative filtering utilizes the collective preferences of a user community to identify patterns and similarities in user behaviors, leading to accurate recommendations. This algorithm constructs a user-item interaction matrix and identifies similar users to make recommendations based on their past interactions. It can be memory-based or model-based, with the latter employing machine learning algorithms to create predictive models. These algorithms have proven effective in domains like e-commerce, movies, and social media, where explicit item attributes may be limited. In this project, we leveraged the ListenBrainz dataset to build two models: the baseline popularity model and the ALS model, for performing recommendations. To enhance the performance of our models, we incorporated LightFM and evaluated the impact of these improvements. By comparing the results before and after incorporating LightFM, we were able to assess the effectiveness of our enhancements.

2 DATASET AND DATA PREPROCESSING

2.1 ListenBrainz Dataset

The ListenBrainz Dataset consists of 21,684 users and a vast number of 695,032,775 playtime records. The dataset is divided into three tables: tracks, interactions, and users. For our models, we focused on the tracks and interactions datasets. These datasets were divided into train and test sets, with the train set containing data from 2015 to 2017 and the test set containing data exclusively from 2019. In our case, we specifically utilized smaller versions of the dataset that contained records solely from 2018.

2.2 Data Preprocessing

The datasets used in both models are 'Interactions' and 'Tracks'. After loading these two datasets to Python, we joined them based on user_id. We then replaced recording_mbid for each row with recording_msid if recording_mbid is 'None' and renamed this column as 'itemID'. This is to ensure tracks with different recording_msid but the same recording_mbid can be counted together. If both recording_msid and recording_mbid are missing, we fill the missing itemID value simply with the corresponding track_name. We also filtered out tracks that have less than 10 plays in total to reduce the dataset size. Finally, we converted itemID from strings to distinct integers, which is more convenient to use for the ALS Model.

2.3 Train/Validation Partitioning

To avoid the cold start problem and to guarantee that all users are present in the training set, we used Window functions to partition the dataset based on userID. This means rows with the same userID will be listed together and form one partition. After that, we employed percent_rank method to rank the entries in each partition based on their timestamps. We then split our dataset by putting the first 70% of each partition into the training set and the last 30% into the validation set.

3 MODEL IMPLEMENTATION

3.1 Evaluation Metrics

We used the following three metrics for evaluation in order to get a better picture of how our model performs. For the Baseline Popularity Model, we only adopted mAP(Mean Accuracy Precision) for evaluation, while for the ALS model, we used all three.

- Mean Accuracy Precision: mAP takes into account the order or ranking of the recommended items. mAP considers not only whether the recommended items are relevant, but also the position at which the relevant items appear in the ranked list
- Precision: the number of correctly recommended items divided by the total number of recommended items. It represents how well the algorithm selects items that the user would find useful or interesting
- RMSE: the root mean squared error of the ALS model on train set. It indicates the difference between the actual 'rating' (in our case the frequency) and the predicted one. It serves as a reference of how well the model fits the training data.

3.2 Baseline Popularity Model Implementation and Evaluation

We implement our baseline popularity model according to the following formula.

$$P[i] \leftarrow (\textstyle\sum_u R[u,i]) \;/\; (\,|R[:,i]|\, + \beta)$$

The i in our target $P[i]$ represents a combination of one user and a track they listen to. So for each track, its $P[i]$ values for different listeners will be different. We group by 'userID' and 'itemID' to count the frequency of a user listening to a track. This will be the values for the $|R[:,i]|$ variable in the denominator. $\sum_u R[u,i]$ will be the total plays of each track among all users. β is the damping

Table 1: Mean Average Precision on Validation set for Different β

β	MAP
0	0.01666
10	0.01617
100	0.02189
500	0.03744
1000	0.04429
5000	0.05136
10000	0.05146
15000	0.05163
20000	0.05181

Table 2: Mean Average Precision Evaluation for the Test Set

β	Test MAP
5000	0.586784

factor we are going to tune. We sort the resulting $P[i]$ values in descending order to get our popularity predictions.

The ground truth we adopted for the baseline popularity model is the top 100 most streamed tracks in the validation set. We count the total plays of each track in validation and sort them in descending order to find the top 100. Since we partition train and test sets according to timestamp of each user, we would like to test how well our baseline model matches the later overall listening trend.

We tune β by evaluating the mean average precision (MAP) of the popularity ranking produced by each β value compared to the ground truth ranking. The table 1 shows the performance of different β on the validation set.

Observe that as we increase the β value, the mean average precision values for validation set increase. The MAP increases quickly as we increase β at first, but no longer makes significant differences after β exceeds 5000. So we just select $\beta = 5000$ for our model.

We apply this model with damping factor $\beta = 5000$ to the test set. The computation is done similarly to train and test sets. We apply the baseline popularity model to the former 70% data on the test set and compare the ranked top 100 to the most streamed tracks in the later 30% data. The resulting MAP is 0.586784.

3.3 ALS Model Implementation

PySpark alternating least squares (ALS) that learns latent factor representations for users and items is implemented as our recommendation system. The ALS prototype analyzes user, item, and the corresponding rating to provide recommendations. In our case, the items are the tracks, uniquely identified by the 'int_itemID' column; the users are the users who listen to the tracks, uniquely identified by the 'userID' column; and the "ratings" are the listen counts (frequency) of each track by the users. Since the feedback is track listen counts (frequency), unlike movie ratings, they are implicit feedback, so the 'implicitPrefs' parameter of our ALS model is set to be True.

Table 3: Mean Average Precision Evaluation for the Test Set

rank	regParam	alpha	Test MAP
200	0.1	1	0.05311

The ALS model is trained using the integrated train set, with the listening frequency of each user with each track. We then use the ALS recommendForAllUsers() function to recommend 100 songs for each user individually, so the recommendations will be unique to each user. We choose the tracks listened by each user from the validation set to be our ground truth for recommendation evaluation and parameter tuning. Intuitively, we evaluate whether the recommended tracks correspond to the tracks each user indeed listen to in a later time, as we partition the train and validation set based on each user's timestamps. If more recommended tracks for a user are indeed played by him/her, we consider the recommendation system better. We conduct such evaluation by computing the mean average precision (MAP) of the recommendations vs ground truth.

The parameters being tuned are **rank**, **regParam**, and **alpha**. **rank** is the number of latent factors in the ALS model, and we tested several values started from the default 10 to 200. **regParam** is the regularization parameter, and we tested [0.01, 0.1, 1]. **alpha** is the implicit feedback parameter, we tested 1 and 10. The parameter tuning history is in table 4. Generally, **rank** is the most significant factor affecting the validation mean average precision. The MAP increases as rank increases. For **regParam**, 0.1 performs better than the other two values. For **alpha**, 1 performs better than 10 generally. In the combinations of these three parameters we have tested, **rank = 200**, **regParam = 0.1**, and **alpha = 1** perform the best, with the validation MAP to be 0.061372, and precision at 100 to be 0.129533. The RMSE on train set is also smaller than the others. Observe that the improvement in the MAP becomes smaller (holding regParam = 0.1 and alpha = 1) as rank becomes larger than 100. Thus, even though continuing to increase rank to be larger than 200 might still improve the MAP, the tradeoff between the possible minor MAP improvement and the training time has us stay with rank = 200 for our model.

We apply this optimal ALS model to test data, and make recommendations for test set users. The tracks listened by each user in test set are used as the ground truth. The resulting MAP for test recommendations is 0.05311.

4 DOCUMENTATION OF EXTENSION: LIGHTFM

4.1 Implementation

LightFM is a Python library that provides efficient recommendation algorithms for handling implicit and explicit feedback. It incorporates item and user metadata, enabling recommendations for new items and users. While Spark offers distributed computing and scalable machine learning, LightFM focuses on simplicity and the integration of metadata for improved recommendations. In this project, we focused on using the LightFM package to improve the performance of our collaborative filtering algorithm.

Table 4: ALS Model Performance

rank	regParam	alpha	Val meanAveragePrecisionAt(100)	Val precisionAt(100)	Train RMSE	Training Time (seconds)
10	1	1	0.002553	0.015194	6.413948	778.25
50	0.1	1	0.034230	0.088710	6.367133	1971.81
50	1	1	0.016180	0.001455	6.3946337	1550.66
75	0.1	1	0.041072	0.100458	6.355844	4005.45
75	0.1	10	0.028050	0.081001	6.270787	2287.54
75	0.01	1	0.038800	0.096807	6.351051	3434.58
85	0.1	1	0.042829	0.103394	6.351660	5689.98
100	0.1	1	0.047015	0.109421	6.346762	2966.21
100	0.1	10	0.035799	0.093608	6.259968	2076.05
125	0.1	1	0.050831	0.114480	6.339419	3068.97
150	0.1	1	0.057529	0.123224	6.333316	5007.57
150	1	1	0.051596	0.095030	6.375749	6492.69
200	**0.1**	**1**	**0.061372**	0.129533	6.322614	8265.87

Table 5: LightFM Model on Validation Set

no_component	*user_alpha*	*precision_at_*100
50	0.01	0.22893
50	0.1	0.24902
75	0.01	0.22764
75	0.1	0.24831
100	0.01	0.22908
100	0.1	0.25622
150	0.01	0.23154
150	0.1	0.25686
200	0.01	0.23558
200	0.1	0.26012
300	0.01	0.23420
300	0.1	0.26239

Table 6: Average Running time

user_alpha	*AverageRunningTime*
0.01	338
0.1	350

Table 7: LightFM Model on Test Set with Chosen Parameters

no_component	*user_alpha*	*precision_at_*100
300	0.1	0.3226

Since we already partitioned our dataset into train and validation. We identified userID, itemID, and interactions(timestamp) in the dataframe, and convert them into a COO matrix that can fit into LightFM function in Python. Then, we trained the LightFM model on the dataset, choosing suitable hyperparameters(rank and regPara) based on what we selected for the ALS model. We performed experiments using the Weighted Approximate Rank Pairwise Loss Personalized Ranking algorithm. We also evaluated the precision at k 100 for the test set, calculating the mean score. This metric was then compared with the ALS MAP Ranking metric to assess performance. Based on validation result, we chose rank(no_component) = 300 and regparam(user_alpha) = 0.1 and applied it to the test set.

4.2 Performance Comparison

Because of the limited resource on HPC, we are unable to run the models on the large dataset. Thus, we scaled down the dataset size and performed all functions on the small dataset. In our case, LightFM is faster and more ideal in making recommendations than the ALS model. Given similar hyperparameters, LightFM achieves slightly higher precision value than ALS model, and the training requires much less time. The results might be different if they are applied to the large dataset, as ALS model is theoretically better at handling large datasets than single-machine LightFM.

5 REFERENCES

[1] LightFM Documentation
(https://making.lyst.com/lightfm/docs/quickstart.html)