Recommendation System on the ListenBrainz dataset

Aarav Pandya[ap7641], Pradhyumn Bhale[pb2777]
https://github.com/nyu-big-data/final-project-group-14/

1 INTRODUCTION

Recommendation systems [1] [2] are essential for delivering personalized song suggestions on platforms like Spotify and YouTube Music. In applications like Music retrieval system, where recommendations play a big role [3], it is important to use enhanced content-based similarity by learning from a subset of collaborative filter data, addressing the limitations of collaborative filtering in music recommendation tasks, particularly for new or less popular items. Various algorithms underpin recommendation systems, including popularity-based recommendations, collaborative filtering, and content-based filtering. Popularity-based recommendations emphasize an item's popularity, such as the number of listens a song receives or the unique users that have engaged with it. In this report, we explore our implementation of a popularity-based baseline model, a latent-factor model using Alternating-Least Squares (ALS) for song recommendations, utilizing the ListenBrainz dataset to create a tailored recommender system. We also implement a single-machine implementation of the LightFM model and benchmark the comparison of the training time and precision accuracy for the ALS and LightFM implementations.

2 SPLITTING DATA AND PREPROCESSING

2.1 Preprocessing

The ListenBrainz platform gathers interaction data from various sources, each possessing distinct identifiers (IDs). To address this, ListenBrainz initially creates a MessyBrainz Identifier (MSID) based on metadata within the recordings dataset. Subsequently, ListenBrainz periodically consolidates multiple MSIDs representing the same song into a single MusicBrainz Identifier (MBID).

For the preprocessing of our datasets, we merged the interactions dataset with the recordings dataset, replacing the MSID with the corresponding MBID when available, and retaining the MSID when an MBID was absent. This process streamlines the dataset, enhancing its stability and reducing its size for more efficient analysis. Additionally, we excluded songs with fewer than 10 unique users, as informed by the distribution analysis in Figure 1. We also show some unfiltered and unprocessed statistics of the dataset in Table 1.

Metric	mean	stddev	min	max
users_per_rec	7.51	40.34	1	7549
unique_users_per_rec	2.54	8.28	1	964
interactions_per_user	22691.37	36651.96	1	1316412
unique_recs_per_user	7701.66	15110.85	1	937299

Table 1: Raw Data Statistics

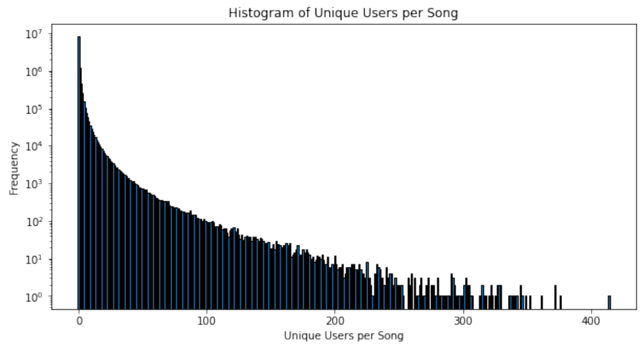

Figure 1: Histogram of Unique users per song

2.2 Train, Validation and Test Split

We explored various split strategies for our dataset. Initially, we divided it into train and validation sets based on user IDs, but this did not reflect real-world conditions. Instead, we opted for a timestamp-based split, allocating 80% of interactions to the train set and 20% to the validation set for each user. This better mimics the use of historical data to predict future behavior. For the ALS model implementation, we used a StringIndexer to have an floating-point index for the recording ids. The test set was given to us in a separate file but it was essential that the string indexing on both the training and test dataset is consistent. For this, we first joined the train and test dataset during our ALS model implementation. Then, we used the previously defined split on timestamp function to split the datasets on the appropriate time stamp to get the test dataset as a dataframe.

3 BASELINE MODEL IMPLEMENTATION

In building our popularity baseline model, we first calculate the global bias, defined as the total number of interactions divided by the total number of songs, representing the average number of interactions per song. Next, we compute the average interactions per user for each song by dividing the total interactions minus the global bias by the number of users plus damping, then normalize this score, which represents the item bias.

Additionally, we determine the unique number of users who interacted with each song, removing bias and normalizing the results. We perform a weighted sum of these two scores, assigning weights between 0 and 1 such that their sum equals 1. The weight for the interaction score can be obtained by $1 - user\_weight$, which is specified in the table. This approach addresses two essential criteria for evaluating the overall performance of our baseline popularity model. We proceed to define our evaluation criteria and present the results.

4 LATENT FACTOR MODEL - ALS

The Alternating Least Squares (ALS) based latent factor method [4] [5] is an algorithm for making personalized recommendations in recommender systems. It is particularly useful for large-scale collaborative filtering tasks, where a system predicts users' preferences based on their historical behavior or interactions with items. In this context, ALS is used to solve the matrix factorization problem, which aims to discover latent factors that explain the observed interactions between users and items.

4.1 Hyperparameters

Utilizing the ALS module from Python's PySpark ML library, we develop and train our models. Our objective is to generate recommendations for each user's top 100 items from the entire item pool. The initial evaluation involves comparing these recommendations with the validation set's ground truth. Subsequently, we juxtapose them with the actual test dataset provided. To configure the dataframe columns in line with the ALS model's prerequisites, we start by calculating the total count of user-song interactions for every unique user and recording combination. This value is normalized against each user's total interaction count to derive an average rating for every song. The model's fine-tuning involves two hyperparameters: *rank*, denoting the quantity of user/item latent factors, and the regularization parameter, labeled *reg*.

5 EVALUATION CRITERIA

In assessing both our models - Baseline popularity based model and the ALS- Latent Factor Model, we utilize two key metrics: precision and Mean Average Precision (MAP). Precision is used to calculate the ratio of pertinent instances within the instances that have been retrieved, while 'Precision at k' quantifies the accuracy of the model's predictions within the top 'k' user recommendations. However, precision@100 overlooks the sequence of recommendations, which is not ideal. On the other hand, MAP takes into account the recommendation sequence, offering rewards to the model for accurate recommendations positioned at the top of the list. This metric provides a comprehensive evaluation, taking into consideration both the users' preferences and the need for ranking relevant recommendations. Consequently, precision@100 supersedes MAP since it does not consider the recommendation order strategy.

6 RESULTS- BASELINE MODEL

We tabulate the results for the Baseline-popularity based model on our training dataset, validation dataset (both derived from the large dataset) and the test dataset in Tables 2, 3 and 4 based on the different baseline model hyperparameters described in previous sections.

β	Split	user_weight	precision@100	mAP
3000	80-20	0.3	0.0071	0.0150
5000	80-20	0.5	0.0058	0.0158

Table 2: Baseline Popularity Evaluation Metrics on Training

β	Split	user_weight	precision@100	mAP
1000	80-20	0.2	0.00208	0.0009
3000	80-20	0.3	0.0052	0.0027
5000	80-20	0.5	0.0050	0.0028

Table 3: Baseline Popularity Evaluation Metrics on Validation

β	Split	user_weight	precision@100	mAP
1000	80-20	0.2	0.00208	0.0009
3000	80-20	0.3	0.0065	0.0030
5000	80-20	0.5	0.0065	0.0030

Table 4: Baseline Popularity Evaluation Metrics on Test

7 RESULTS- ALS

We tabulate the results for the validation set (20% split on large dataset) in Table 4 and the test dataset in Table 5 for different hyperparameters for the ALS model as detailed in previous sections.

reg	Rank	precision@100	mAP	ndcgAt(100)
0.01	10	0.00595	0.0004675	0.0074895
0.01	20	0.006	0.00061	0.008
0.01	50	0.00745	0.00087	0.0103
0.01	100	0.00802	0.001	0.01117
0.01	200	0.01044	0.00138	0.0145
0.01	300	0.01146	**0.001597**	0.0159
0.1	10	0.02176	0.00039	0.02422
0.1	20	0.019188	0.0002366	0.0197798
0.1	50	0.019635	0.00024058	0.0201797
0.1	100	0.0209	0.000336	0.022842
0.1	200	0.020489	0.000253	0.021026
0.1	300	0.0049427	0.000248	0.00553
0.5	10	0.02167	0.000346	0.024083
0.5	20	0.02175	0.000337	0.02377
0.5	50	**0.02188**	0.000351	**0.02436**
0.5	100	0.02153	0.00034	0.02375
0.5	200	0.0223	0.0003	0.02275

Table 5: ALS Evaluation Metrics on Validation

reg	Rank	precision@100	mAP	ndcgAt(100)
0.1	10	0.024321	0.00043157	0.028192
0.1	20	0.02145	0.00026	0.023
0.01	300	0.0128	0.00177	0.0185

Table 6: ALS Evaluation Metrics on Test

8 EXTENSION - LIGHTFM

We implemented the single machine implementation of LightFM on our dataset. LightFM is a Python library for building hybrid recommender systems, which can make recommendations using collaborative filtering, content-based methods, or a combination of both. While handling both these scenarios, it also supports hybrid models, where both item and user metadata are used in conjunction with collaborative filtering. One key feature of LightFM is that it's capable of handling cold-start problem, where it's difficult to make recommendations due to a lack of past user-item interactions, by using item or user metadata to make predictions.

Internally, LightFM [6] uses matrix factorization to learn embeddings for users and items, and can use these embeddings in conjunction with additional metadata to make predictions. It supports both learning-to-rank and rating prediction optimization objectives, and allows for both WARP (Weighted Approximate-Rank Pairwise) and BPR (Bayesian Personalized Ranking) loss functions, among others. For our implementation, we use the WARP loss function.

The interaction data between users and songs, represented within a dataframe, undergoes transformation into a sparse matrix format. This transformation is executed post the application of String Indexer on the recording id. Subsequently, the LightFM model is put through the training process using the allocated training split and employing the WARP loss function. Interaction ratios are computed as the metric for the matrix ratings, specifically by dividing the count of interactions between a user and a song by the total interactions of that user. This approach proves advantageous, especially when the primary goal is to optimize the upper section of the recommendation list, often referred to as precision@k. We show the results for precision@100 metric for the large dataset compared with ALS as well as the training time in seconds for the two models in Table 7.

model	Rank	precision@100	Time (s)
LightFM	10	0.01407	76.34
LightFM	300	0.02439	290.44
ALS (in Spark)	10	0.02176	38.99
ALS (in Spark)	300	0.01156	3164.91

Table 7: Benchmarking between LightFM and ALS

9 CONTRIBUTIONS

Aarav Pandya: Partitioning of data, Baseline Model, ALS Model, benchmarking the LightFM extension and their documentation.
Pradhyumn Bhale: Partitioning of data, Baseline Model, Hyperparameter Tuning for ALS, LightFM Model and their documentation.

REFERENCES

[1] Robin Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12, 11 2002.

[2] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009, jan 2009.

[3] Brian McFee, Luke Barrington, and Gert Lanckriet. Learning content similarity for music recommendation. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(8):2207–2218, 2012.

[4] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[5] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE International Conference on Data Mining*, pages 263–272, 2008.

[6] Maciej Kula. Metadata embeddings for user and item cold-start recommendations. *CoRR*, abs/1507.08439, 2015.