An Implicit Feedback Music Recommender System

Big Data Capstone Project

Harsha Koneru
hk3820@nyu.edu

Sharad Dargan
sd5251@nyu.edu

Sargun Nagpal
sn3250@nyu.edu

ABSTRACT

In this project, we used a 11.6 GB real-world dataset of music listening behavior containing data on ~8000 users, over 23 million tracks and 179 million user-track interactions to build an implicit feedback music recommender system using Spark.

We preprocessed and partitioned our data, implemented a baseline and Latent Factor model, and evaluated the model recommendations. We compared the multi-node performance with a single machine implementation, and investigated ways to accelerate inference using approximate search. The ALS model achieved a MAP@100 of 0.064 and NDCG@100 of 0.15 on the test set.

METHODOLOGY

1. Data Preprocessing

Since the songs in our dataset need to map to distinct IDs, we first produced unique IDs for each track. We removed duplicate interactions (~15k) in the interactions table. Over 45% tracks only had a single interaction, therefore, we did not remove tracks with few interactions. We also did not remove tracks with a very large number of interactions, since this would bias the results of our popularity model. Furthermore, for the ALS model, we performed hyper-parameter tuning on the count importance, thus invalidating the removal of tracks with high interaction count. On the user level, we did not clip users with interaction counts at the ends, since we define our popularity model based on the number of distinct users per track.

2. Cross-Validation split

We compared several data splitting strategies for recommender systems[1]. Although the Temporal Global split is the most strict and realistic setting, it reduces the number of users and tracks in the train set and aggravates the cold-start problem. Therefore, we used a Temporal User 80:20 split to partition each user's data into the Train and Validation set based on the interaction timestamp. This strategy is better than a random split since it accounts for the time of interaction for each user. However, it suffers from the data leakage problem since the split boundary for each user is not uniform. *Table 1* shows the statistics of our data.

Dataset	Users	Tracks	Interactions
Train	7852	21.1 M	143.5 M
Val	7909	9.0 M	35.9 M
Test	7125	1.9 M	50.0 M

Table 1: **Data Characteristics.**

3. Modeling

3.1 Baseline Popularity Model

We tried three approaches to define track popularity for our baseline model: Number of interactions per user, Number of distinct users, and Number of interactions per track[2]. We used a damping factor β for the interactions per user model, and performed hyperparameter tuning on the validation set with β values 1, 10, 50, 200, 500, 1000, and 10000.

3.1.1 Evaluation

We calculated the Precision@100, which gives a measure of the proportion of correct predictions; MAP@100, which rewards the top ranking of correct predictions; and NDCG@100, which discounts recommendation relevance based on the rank and normalizes the resulting discounted cumulative gain. *Table 2* shows the results of the baseline model.

Evaluation Dataset	Popularity Metric	Damping Factor	MAP@100	NDCG@100
Validation Set	# Interactions/user	β=1	0.000012	0.00011
		β=10	0.000018	0.00023
		β=50	0.00004	0.0005
		β=200	0.00011	0.0021
		β=500	0.00106	0.01173
		β=1000	0.00306	0.02158
		β=10000	**0.00485**	**0.02853**
	# Distinct Users	**-**	**0.00893**	**0.03697**
	# Interactions	-	0.00508	0.02913

Test Set	**# Interactions/ user**	**β=10000**	**0.00098**	**0.00871**
	# Distinct Users	-	0.00086	0.00815
	# Interactions	-	0.00097	0.00869

Table 2: **Results of the Baseline popularity model.**

Baseline model with popularity metric as Number of interactions per user performs the best on the test set (with β=10000 as tuned using the validation set). Note that with high values of beta, this metric closely approximates the number of interactions per track.

3.2 Latent Factor Model

In order to create personalized recommendations for users using collaborative filtering techniques, we employ latent factor models. These models enable us to learn a condensed representation of users and items in a low-dimensional latent space based on implicit feedback data from past user-item interactions. By estimating these latent factors, the model can predict the missing feedback in the utility matrix for each user-item combination, which we use for the recommendations.

3.2.1 Alternating Least Squares

Due to the scale of the data, we trained the model using the pyspark.mllib.recommendation (Spark v3.1.2) implementation of alternating least squares (ALS) on the NYU Dataproc environment, while using HDFS for storing the intermediate files, results and model checkpoints.

3.2.2 Model Selection

To perform hyper-parameter tuning for the latent factor model, we tune 3 parameters - alpha (Counts weights for implicit feedback), regparam (regularization parameter) and rank (dimension of latent representation) by varying them individually, while keeping the other parameters fixed to establish the trend. Then, we narrow the search space for the hyperparameters, for the higher rank models. All the tuning is done on a validation set generated as discussed in Section 2.

From *Figure 1a*, It is clear that alpha of 0.5 achieves the best result, with a marginal improvement over alpha of 0.8 and very bad performance at alpha 1.2. This suggests that with a very high weights to the implicit feedback (interactions count), we are encouraging the model to adjust the factors to account for very high-magnitude counts, which results in overfitting on the training-set and poor generalization performance on the validation set.

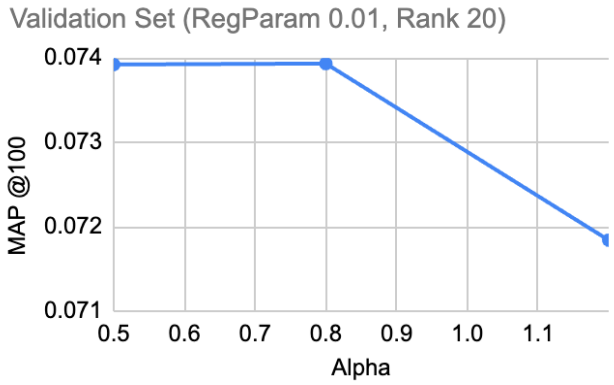

Figure 1a: **MAP as a function of alpha with fixed Rank 20, Regparam 0.01**

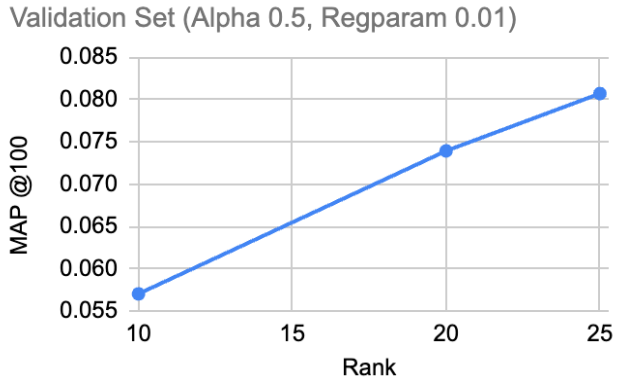

Figure 1b: **MAP as a function of rank with fixed alpha 0.5, Regparam 0.01**

Rank defines the dimension of the latent factor representations, and a higher dimensional representation implies increased model complexity. It is evident that a higher rank is able to give a better performance in terms of MAP@100 on the validation set. The trend in *Figure 1b* suggests that we can increase the model complexity to a greater extent by increasing the rank, but limited by the cluster resource limitations, we stopped at rank 25 (which gave the best MAP).

We also saw that for models with low complexity (Rank 10-20), the optimal regularization parameter was 0.01, but for higher model complexity (Rank 25), a higher regularization parameter 0.1 gave optimal results.

A complete table of hyperparameter tuning results is given in *Table 3*. The best performing model has rank 25, alpha 0.5 and reg-param as 0.1 with a MAP@100 of 0.08234.

3.2.3 **Evaluation**

Rank	Reg-param	Alpha	Precision @100	MAP @100	NDCG @100
10	0.1	0.1	0.11932	0.04919	0.12686
10	0.1	0.5	0.13226	0.05594	0.14083
10	0.1	0.8	0.13332	0.05600	0.14159
10	0.01	0.5	0.13334	0.05703	0.14256
10	0.001	0.5	0.12905	0.05446	0.13815
20	0.01	0.5	0.15982	0.07392	0.1704
20	0.1	0.5	0.15777	0.07263	0.16898
20	0.01	0.8	0.15986	0.07393	0.17093
20	0.01	1.2	0.16088	0.07184	0.16986
25	0.01	0.5	0.17005	0.08069	0.18132
25	0.01	0.8	0.17177	0.08039	0.1823
25	**0.1**	**0.5**	**0.17145**	**0.08234**	**0.18234**

Table 3: **ALS Hyperparameter Tuning Results**

The test-set performance for the best-performing model from the previous section is slightly worse than the validation-set (0.064 v/s 0.082). A potential reason for the dip in performance is that the test is set far ahead in time compared to the validation dataset, which will result in a distribution shift in the data.

Rank	Reg-param	Alpha	Precision @100	MAP @100	NDCG @100
25	0.1	0.5	0.13848	0.063959	0.14834

Table 4: **ALS Test Set Results**

3.2.4 **Comparison with Baseline**

The test-set MAP@100 for the ALS model (0.06396) is a substantial improvement (almost 65x) over the baseline model (0.00098). The improvement is majorly driven due to the personalization of recommendations for the users (long tail).

4. **Extensions**

4.1 **Single Machine Implementation**

LightFM is a python-based hybrid recommender system library that combines both content based and collaborative filtering, offering flexibility and accuracy in recommendation tasks. It is designed to run efficiently on a single machine, unlike Spark which is designed for parallel computing. The goal of this extension is to compare the time-efficiency and performance of LightFM and Spark ALS models on different dataset sizes.

4.1.1 **Technical Details**

We installed LightFM version 1.16 on a singularity container in the Greene cluster. The computation was performed on a single node with 8 cores and 64GB of memory, enabling efficient data handling and processing within the available resources. PySpark (version 3.2.1) was also installed locally in the container in order to keep the evaluation methodology the same as Spark ALS.

The LightFM modules were trained using 'warp' loss[3], which is a ranking-based loss function that optimizes the model by maximizing the margin between positive and negative interactions, improving the quality of the recommendations. LightFM provides options to tune the user and item regularization parameter (called alpha in LightFM), but for the sake of simplicity, and to keep the comparison similar, we always kept both the regularization parameters the same.

For evaluation, the inbuilt functions user and item representations and biases were extracted, and the score corresponding to each user-item pair was computed using the equation $s(u, i) = < w_u, w_i > + b_u + b_i$, where w_u, w_i are the weights of the user and item, and b_u, b_i are the biases of the user and item respectively[4].

We compared the time taken for training, and the MAP@100, Precision@100, and NDCG@100 for the models trained using LightFM on three dataset sizes (20%, 50%, full) and compared it with equivalent models trained using Spark's ALS module. Both models were trained for 10 epochs.

4.1.2 **Hyperparameter Tuning Results**

We initially trained the models with rank = 10, for various values of the regularization parameter. Once we fixed on a regularization parameter, we then increased the number of components to 25 for full training. The results from the hyperparameter tuning are presented in *Table 5*.

The best results are achieved with rank 25 and a regularization parameter of 10^{-6}.

We note that this is not the most ideal way to perform hyperparameter tuning as LightFM provides separate item and user regularization terms, and we are not limited to rank 25 on a single machine as it is not a shared resource with bottlenecks. However, we wanted to keep the variability between LightFM and Spark ALS as similar as possible to be able to compare model efficiency and accuracy effectively, and thus restricted our hyperparameter search.

alpha	rank	MAP @100	Precision @100	NDCG @100
0	10	0.0313421	0.099821	0.101932
10^{-8}	10	0.0322055	0.101652	0.103117
10^{-6}	10	0.0372291	0.112314	0.113815
10^{-4}	10	0.0000035	0.000067	0.000064
10^{-2}	10	0.0000002	0.000013	0.000010
10^{-8}	20	0.0474944	0.134562	0.136681
10^{-8}	25	0.0478640	0.137506	0.139740
10^{-6}	**25**	**0.0702000**	**0.172900**	**0.176100**

Table 5: **LightFM Hyperparameter Tuning Results**

4.1.3 **Results**

Table 6 contains the comparison of the Spark ALS and LightFM models on various dataset sizes. The time taken to train the model on 20% and 50% datasets are comparable. On the full dataset, the time taken by LightFM is much higher at 18 minutes and 23 seconds compared to the 8 minutes and 37 seconds for the ALS model. One key difference between ALS and LightFM is that ALS maintains its performance on both the 20% and 50% datasets, whereas the performance of LightFM severely suffers by reducing the size of the dataset. Note that the dataset size is reduced to 20% and 50% in a stratified manner, where we remove the corresponding proportion of the data for each user instead of randomly removing a portion of the dataset.

Dataset Size	Method	Training Time (min:sec)	MAP @100	Precision @100	NDCG @100
20%	ALS	02 : 14	0.049	0.11502	0.12409
	LightFM	02 : 01	8.34×10^{-6}	0.00001	0.00001
50%	ALS	05 : 24	0.061	0.13359	0.14263
	LightFM	04 : 34	8.44×10^{-6}	0.00016	0.00016
Full	**ALS**	**08 : 37**	**0.064**	**0.13848**	**0.14834**
	LightFM	**18 : 23**	**0.054**	**0.13830**	**0.1404**

Table 6: **Comparison of ALS and LightFM (Test Set)**

4.2 **Approximate Nearest Neighbor search**

For some applications, recommendations must be served real-time and it is crucial to consider the latency of the recommender system. In such situations, the exactness of the recommendations can be sacrificed for accelerated search at query time. In a latent factor model, the recommendations for a user are produced by computing the similarity between the latent representation of the user and each item in the item list, and ranking the items in decreasing order of similarity. This operation has time complexity *O(nd)*, where n is the number of items and d is the latent space dimension. It is possible to accelerate this computation using approximate nearest neighbor search. In this extension, we implemented approximate search and evaluated the efficiency gains and changes induced in accuracy over brute-force search.

4.2.1 **Technical Details**

We used the annoy module[5] (version 1.17.2), which is a C++ library with a Python wrapper developed by Spotify for approximate search. The computation was performed on a single node on the Greene cluster with 8 cores and 128 GB memory. The module implements a forest of binary trees, where each tree recursively partitions the vector space into two regions until there are at most k points in each partition. The points in the same partition are nearest neighbors. At search time, the algorithm also evaluates other neighboring splits to find nearest neighbors using a priority queue[6]. The search time is reduced to *O(log n * d)*.

The algorithm has two important hyperparameters: *n_trees* (number of trees to construct) and *search_k* (number of candidates to find). Increasing the number of trees and candidates leads to a greater probability of finding favorable splits and points that are close to each other in the latent space. The default number of candidates are the product of the number of trees and the number of closest neighbors.

We used ALS with 20 latent factors, which took an inference time of 1 hour, 30 minutes, and 34 seconds. The metrics for ALS were Precision@100=0.16, MAP@100=0.07, and NDCG@100=0.17. To implement ANN, we need to define an index and add latent representations to the index. This was done using a Dask bag to reduce memory usage. The index is then built based on the number of trees we define.

4.2.2 **Results**

Table 7 shows the results of hyperparameter tuning of the annoy algorithm. For each configuration, adding the latent representations to the index takes about 10 minutes, and further time to build the index (indicated in the table). As expected, if we increase the number of trees, the metrics on the validation set improve, but the approximate nearest neighbor search takes a longer time (*Figure 2*). Increasing the number of candidates (k) leads to an increase in the metrics with little to no increase in the search time. The best results were obtained with 50 trees, 500k candidates.

This is a 8x (700%) speedup, but just a 17% decrease in the Precision@100. Thus by just working with a small fraction of nearest neighbor candidates (0.5M out of 23M), it is possible to obtain results that are comparable to exhaustive search.

It is worth noting that the time taken by ALS with exhaustive search may be affected by cluster resources and warrants further investigation to ascertain the true speedup under identical conditions.

Hyper-parameter		Time			Metrics		
n_ trees	k	Index build	ANN search	Speed up	Precision @100	MAP @100	NDCG @100
1	100	1m 53s	0.20s	27k	0.0019	0.0002	0.0020
10	1k	2m 36s	1.37s	4k	0.0112	0.0016	0.0111
20	2k	5m 28s	8m 40s	10.5	0.0168	0.0025	0.0160
50	5k	15m 35s	17m 59s	5.0	0.0278	0.0046	0.0277
50	10k	15m 35s	17m 16s	5.0	0.0382	0.0070	0.0384
50	25k	15m 35s	11m 12s	8.1	0.0566	0.0120	0.0571
50	50k	15m 35s	10m 38s	8.5	0.0733	0.0176	0.0737
50	**500k**	**15m 35s**	**11m 24s**	**7.9**	**0.1331**	**0.0470**	**0.1338**
100	10k	27m 1s	20m 10s	4.5	0.0413	0.0077	0.0415
200	20k	61m 9s	98m 17s	0.9	0.0583	0.0123	0.0580

Table 7: **ANN Validation Results**

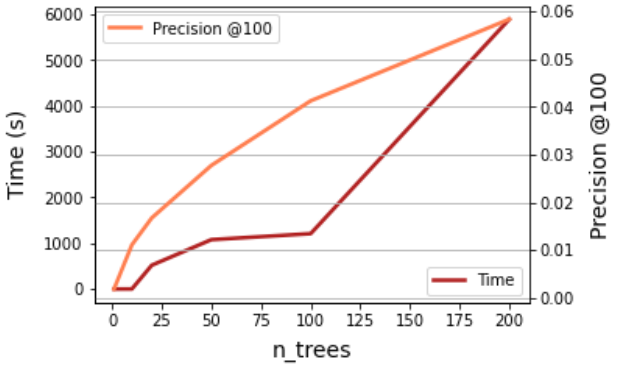

Figure 2: **Number of trees vs precision, search time.**

To conclude, improved precision comes at the cost of increased search time (*Figure 3*). By appropriately tuning the hyperparameters for the approximate search, it is possible to obtain a significant speedup in inference time compared to the ALS model, while still maintaining a similar recommendation performance. Furthermore, performing search on richer user-tem latent representations (higher rank) might mitigate this tradeoff between speed and performance further.

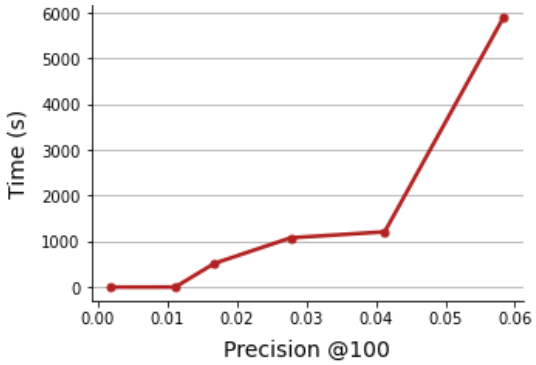

Figure 3: **Search time vs Precision gain**

5. Conclusion

The ALS Latent Factor model exhibited a remarkable 65-fold improvement over the baseline popularity model, indicating its superior recommendation capabilities. The ALS model can be further improved by increasing the complexity through a higher rank model, provided computational resources. While a single machine implementation of LightFM on large datasets took slightly longer than ALS on a multi-node setup, their performance was similar. However, on smaller datasets, ALS maintained better prediction accuracy compared to LightFM, which showed a decline in performance. To increase inference speed, we can leverage Approximate Nearest Neighbors (ANN) techniques like annoy, potentially enhancing the overall efficiency of the recommendation system.

6. Author Contributions

Harsha focused on train-validation split and LightFM model creation, Sargun worked on the Baseline model and approximate nearest neighbor search, while Sharad handled data preparation and ALS. All team members reviewed and provided valuable feedback on each other's work, contributing equally to the project.

REFERENCES

[1] Zaiqiao Meng University of Glasgow et al. 2020. Exploring data splitting strategies for the evaluation of recommendation models: Proceedings of the 14th ACM conference on recommender systems. (September 2020). Retrieved April 26, 2023 from https://dl.acm.org/doi/10.1145/3383313.3418479

[2] Yitong Ji Nanyang Technological University et al. 2020. A re-visit of the popularity baseline in Recommender Systems: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. (July 2020). Retrieved April 26, 2023 from https://dl.acm.org/doi/10.1145/3397271.3401233

[3] Weston, Jason, et al. "Learning to Rank Recommendations with the K-Order Statistic Loss." Conference on Recommender Systems, 12 Oct. 2013, https://doi.org/10.1145/2507157.2507210. Accessed 27 Apr. 2023.

[4] Maciej, Kula, and Lyst. Metadata Embeddings for User and Item Cold-Start Recommendations. https://arxiv.org/pdf/1507.08439.pdf

[5] Spotify. "Spotify/Annoy: Approximate Nearest Neighbors in C++/Python Optimized for Memory Usage and Loading/Saving to Disk." *GitHub*, github.com/spotify/annoy. Accessed 16 May 2023.

[6] Erik Bernhardsson. "Nearest Neighbors and Vector Models – Part 2 – Algorithms and Data Structures." *Erik Bernhardsson*, 19 Apr. 2020, erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html.