Recommender System Final Report

Jin Choi
Center For Data Science
New York University
New York, New York
jkc9890@nyu.edu

Abstract

This project presents the development and evaluation of a collaborative filtering recommender system leveraging the Alternating Least Squares (ALS) model. The model is designed to recommend songs to users based on implicit feedback (the count of songs listened to) for each user-item pair. The source of the data is from ListenBrainz and it is processed using the NYU High Performance Computing (HPC) dataproc cluster. The ALS model's recommendations are compared and evaluated against a popularity baseline model, which suggests the most listened to songs to all users. The performance of both models is assessed using the Mean Average Precision at K (MAP@K) metric, providing a robust measure of the quality of the ranked lists of recommendations.

1. Data Preprocessing

During the pre-processing phase, we began by checking for missing data or irrelevant columns. However, these datasets were already cleaned, so no further cleaning was necessary. Then we concentrated on exploring the variables in the track and interactions datasets to identify the 'key' value required for further steps. Our findings showed that while a song could have multiple recording_msid assigned, recording_mbid is an unique identifier (unless when null). Therefore, if a song contained a recording_mbid, we replaced the recording_msid of the song to recording_mbid. We filtered out user_ids with less than 10 unique recording_msid, and also filtered out recording_msids with less than 10 unique user_id. This is to minimize the noise in the data and only keep the data with sufficient information.

1.1 Data Partitioning

For partitioning, our goal was to partition the dataset into a train and validation set, with a split ratio of 8:2. However, a simple random split wouldn't have allowed us to ensure that all user-interactions are in the training set. To address this, we incorporated a user-based partition where we create a distinct list of ‘recording_msid’ for each user and split each user’s interaction into 8 to 2 ratio, where 80% of the interaction goes to the training set and the remaining goes into the validation set. This process ensures that every user in the training set is in the validation set while preventing the interaction to overlap in training and validation.[1] This avoids the cold start problem where the validation set does not have a user’s history.

2. Evaluation Metric

For this project, we utilized the Mean Average Precision at 100 (MAP@100) as our primary evaluation metric. This decision was largely influenced by the nature of our recommendation task. MAP@100 is a ranking-based metric, focusing on the order of recommended items rather than the precise prediction of individual preferences. It provides a single summary number of the quality of ranked lists of recommended items, where the '100' indicates we are looking at the top 100 items recommended. This aligns well with our goal of recommending the top 100 songs.

3. Popularity Baseline Model

3.1 Model Implementation

The popularity baseline model serves as a simple, efficient tool for recommendation systems, providing a benchmark for more complex models and effectively addressing the 'cold start' problem. We devised a baseline popularity model as follows:

$$C * log(\text{'}Distinct\ userid\text{'} + 1) + (1 - C) * log(\text{'}Total\ interaction\text{'} + 1)$$

This model calculates a popularity score for each item (in this case, each recording_msid) based on a combination of the number of distinct users who interacted with the item and the total number of interactions the item received. The formula uses a logarithmic transformation to reduce the influence of a few users who might have interacted with an item multiple times and inflate its popularity. The C hyperparameter adjusts the relative importance of the total interactions compared to the distinct users' count in the final score. The C hyperparameter is in the range of 0 to 1 in the increment of 0.1, and for each C value, we calculate the MAP and find the value C that outputs the highest MAP for the validation set.[2] From hyperparameter tuning, the highest MAP resulted when **C = 0.9**. This meant that adding some proportion of total interaction improved the model performance.

3.2 Model Evaluation

Using the MAP@100 metrics, the results for the small and full dataset were as follows:

Result		
	Validation	Test
Small	0.0004942	0.0009574
Full	0.0004317	0.0000462

Figure 1. Popularity baseline model evaluation

Overall, our baseline model's performance was low across all small and full sets, highlighting the lack of necessary individualization for each user and the need for improvement. However, this evaluation provided valuable insights into the model's limitations. Specifically, the model performed better on the validation sets than on the test set, indicating that our model's recommendations may not fully reflect the personalized needs of general users. Furthermore, it performed better on small datasets than on the full datasets. That being said, these results suggest that our baseline model has significant room for improvement in accurately predicting user-item interactions. Nevertheless, it sets a fundamental baseline for the development of our ALS models, which we will elaborate on in the subsequent section.

4. Alternating Least Squares (ALS) Model

4.1 Model Implementation

We employed the parallel Alternating Least Squares (ALS) algorithm from Spark's machine learning library in our recommendation model (*pyspark.ml)*. This algorithm was applied to our training data to decipher latent factor representations for both users (user_id) and items (recording_msid) These representations facilitated prediction of user preferences, enabling us to generate the top 100 item recommendations for each user.

As the ALS model takes integer indices as input, we converted the 'recording_msid' column to an integer index using PySpark's module *'StringIndexer'*. We fitted the indexer to the training data and transformed the training, validation, and test data accordingly.

4.2 Hyperparameter Tuning

To tune the ALS model, our initial approach was to implement a grid search method using PySpark's ParamGridBuilder module. However, due to the high computational demand, this approach resulted in slow processing and led to the termination of the worker processes in the Dataproc cluster.

To circumvent these challenges, we shifted our approach towards manually testing different combinations of hyperparameters. This strategy involved initializing the Alternating Least Squares (ALS) model with various parameter sets and running the model for each combination. Despite being more labor-intensive, this method was more feasible given our computational constraints and ensured that we were able to derive results within a reasonable timeframe.

Specifically, the parameters we adjusted included the **'rank'**, **'maxIter'**, **'regParam'**, and **'alpha'** of the ALS model. The rank parameter determines the number of latent factors in the model, maxIter specifies the maximum number of iterations to run, regParam controls the regularization strength, and alpha adjusts the confidence associated with implicit feedback.

With the full training dataset, the ranges of the hyper parameters that we used to tune our model are as follows:

Range of hyperparameters used						
rank	50		100	150	200	
regParam	.0001	.001	.01	.1	1	2
alpha	5			10		
maxIter	15					

Figure 2. Range of hyperparameters used

The below graphs show the effect of each parameter on the MAP@100 score:

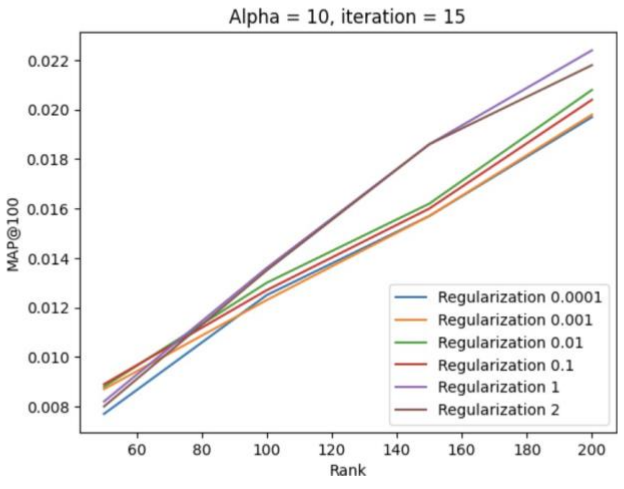


Figure 3. MAP@100 with alpha=10, Iteration=15

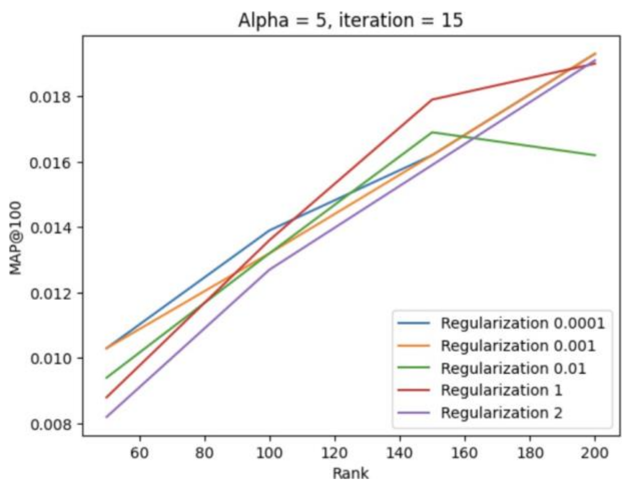


Figure 4 MAP@100 with alpha=5, Iteration=15

Based on the hyperparameter tuning using the small and full validation set trained on small and full train set, the following parameters were chosen:

	rank	maxIter	alpha	regParam
Small	100	15	10	.01
Full	200	15	10	1

Figure 5. Best ALS hyperparameters for validation sets

It is worth noting that as rank increases, the MAP value also increases. However, there is a trade-off between increasing the rank to improve the MAP value and slowing down the computation time for running the model. Further, increasing the rank can also overfit the training data and may not generalize well to the unseen data. Another point is that increasing the maxIter value will keep increasing the MAP score until it converges. Best practice is to find the point of convergence. However, using this approach is computationally expensive and impossible to complete in the Dataproc cluster (due to the killed session). Thus, we kept the maxIter value as 15, which improves the MAP score while balancing the computation time.

4.3 Model Evaluation

Using the aforementioned hyperparameters, the ALS model was trained using the full training data and was evaluated using both the validation set and the test set.

Evaluation Type	Score (MAP@100)
Validation Small	0.01628
Validation Full	0.02194
Test	0.05147

Figure 6. PySpark ALS model evaluation on different sizes

From this result, there is a clear improvement in using the personalized recommender system compared to the popularity baseline model.

5. Extensions

5.1 Single-Machine Implementation (LensKit)

In this segment, we performed a comparative analysis between Spark's parallel ALS model and LensKit, evaluating the computation time and accuracy.

To ensure consistency in the datasets utilized in both Spark's parallel ALS model and LensKit, we imported the segmented datasets that were previously utilized for the implementation of the Spark ALS model. Furthermore, while PySpark's ALS model contains 'coldStartStrategy' parameter to indicate whether the ALS model wants to ignore cold start problems, the ALS model in LensKit does not have such parameter. Therefore, we manually dropped user_ids in the test set that was not present in the training dataset.

For a more accurate comparison, we employed the ALS model offered by LensKit. Specifically, we used the method in the ALS library 'ImplicitMF' to illustrate the user's implicit feedback behavior.

We adhered to the same hyperparameter ranges for model tuning, but we didn't replicate the exact parameter values, acknowledging that PySpark's ALS model and LensKit's ALS model might not be precisely identical. This necessitated a distinct hyperparameter tuning process. After extensive tuning on the complete training set, the optimal MAP@100 on the validation set was found to be:

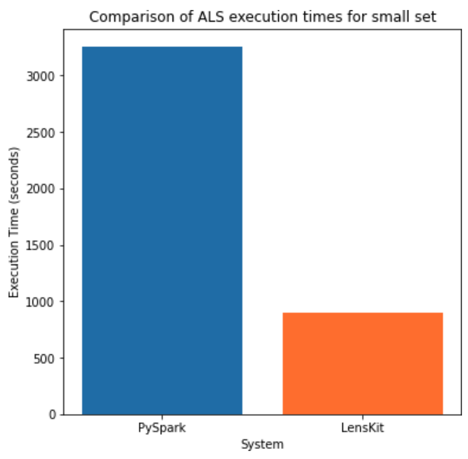
	rank	maxIter	alpha	regParam
Small	150	15	5	.1
Full	200	15	10	1

Figure 7. LensKit's hyperparameters for validation sets

Finally, we calculated the duration required to fit an ALS model across different sizes of the training dataset. For accuracy, we employed MAP@100, to ensure the assessment was coherent with the analysis performed on Spark's ALS model.

5.2 Result

Ideally, the dataset would be broken down into subsets of data and we would measure the time it takes to train the ALS model as a function of file size. Due to the limitation of the cluster, we decided to just test two data sizes for the model efficiency and accuracy. Small set is roughly 2 GB in size, and the full set is about 7.2 GB in size.



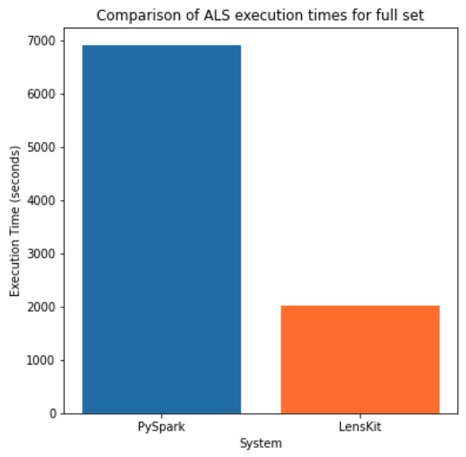


Figure 8. Time comparison for small and full set

As shown above, there is a significant difference in the computation time for PySpark's ALS version versus that of LensKit. As discussed in lecture, single-machine is preferred when the data is between 2 to 20 GB for reading in-memory, which corresponds to the dataset that we are working on. This illustrates that for the ListenBrainz music data that we are currently using, single-machine implementation is preferred in terms of saving computation time. Then, we analyzed the model accuracy and compared it with the PySpark version. The below metric uses MAP@100.

	Validation (small)	Validation (full)	Test
PySpark	0.01628	0.02194	0.05147
LensKit	0.01253	0.02738	0.03748

Figure 9. MAP@100 for LensKit and PySpark's ALS

PySpark's MAP@100 score was higher for both validation (small) and test, while LensKit's MAP@100 was higher for validation (full). From the above table, there is a performance benefit in using PySpark's version compared to the LensKit version. There are several reasons why there could be a difference in the MAP@100 score:

Cold start handling: In PySpark ALS, we set coldStartStrategy="drop" which means that users/items not seen in the training data will be completely dropped from the recommendations. In LensKit, there is no explicit parameter to handle cold-start items, and it was handled manually.

Non-negative factorization: In the PySpark ALS implementation, we have set nonnegative=True, which enforces non-negative matrix factorization. This constraint is not set in LensKit ALS implementation, as it does not support non-negative matrix factorization.

There was no built-in MAP function in LensKit, so we had to build it from scratch. There may be subtle differences in how the evaluator was implemented.

5.3 Item-based Nearest Neighborhood Search

In this section, our focus was on developing a recommendation system using a hybrid approach that combines item-based methods. For users with history, we leveraged Nearest Neighborhood Search to find similar items, while for users without a history, we utilized cosine similarity on metadata. By incorporating this item-based approach, we aim to address the "cold start" problem, which occurs when there is limited or no user history available for accurate recommendations.

5.3.1 Item Based Search for Cosine Similarity on metadata - Users without History

Including the "season" attribute allows for capturing seasonal variations in music popularity, such as the increased popularity of holidays songs. It enhances the relevance and personalization of recommendations by aligning them with the user's current preferences and the context of the season.

The first step involves preparing the data by preprocessing and encoding the categorical features into numerical representations. Next, the cosine similarity matrix is computed, which measures the similarity between each pair of metadata vectors. The higher the cosine similarity score, the more similar the metadata attributes are. Building the Annoy index is then performed, where the metadata vectors are indexed to efficiently retrieve similar items. This index can be utilized to make recommendations by querying for items that are similar to a given item. Although we were unable to proceed with the computation of the cosine similarity matrix due to memory limitations, we believe that the completed steps and the initial implementation of the recommendation system hold promise for delivering valuable recommendations to our users.

5.3.2 Nearest Neighbor Item Based Search - Users with History

For the users with music listening history, a nearest neighbor item-based search model was constructed. BucketedRandomProjectionLSH was used for this purpose. The first step for this approach also involved encoding string values into numerical values. This step involved indexing the 'user_id' and 'recording_msid'. Indexed variables are then transformed into feature vectors to further facilitate the modeling process. The BucketedRandomProjectionLSH model was used to identify nearest neighbors.[4] LSH hashed input items so that similar items map to the same "buckets" with high probability. This made it possible to identify probable nearest neighbors quickly by only comparing items within the same bucket. Then, a pipeline was set up to streamline the entire modeling process. After these preliminary steps, the model was fitted on the training data and the data was subsequently transformed using the newly fitted model. Approximate Nearest Neighbor Search was performed with a key with Vectors.dense(1.0, 0.0).[4]

In the next step, we aim to provide specific song recommendations based on the outputs from the approximate nearest neighbor search. The output was grouped by 'user_id' and sorted within each group based on 'distCol', a column representing distances between the items. A rank column was then added to each group, enabling the identification of top-ranked items within each user group. The rank column was filtered to only keep the top 'k' records for each user retained. User Id and songs were displayed in a list as the final output, providing each user with a personalized set of song recommendations. Finally to evaluate the precision of the model, MAP@100 was employed.

5.4 Result

In the initial stages of the second extension, our strategy was to create bifurcated results for two user categories: users with and without listening history. The plan was to implement a cosine similarity search on metadata for users lacking history, while deploying a nearest neighbor item-based search for users with history. However, during the execution phase, we faced challenges in successfully implementing the cosine similarity search on metadata. Despite exploring various avenues, we were unable to establish a functional code that would yield the desired results. Consequently, the results in the following sections solely represent the outcomes from the nearest neighbor item-based search model, which were applied to the users with history.

To tune the Nearest Neighbor Search model, a manual testing with different combinations of hyperparameters was done. Although it is not the most effective approach to perform hyperparameter tuning, it was the most feasible approach at the moment as the cluster was slowing down as many people were using it at the same time. Parameters adjusted were '**bucketLength'** and **'numHashTable'.** 'BucketLength' is the length of each hash bucket. A larger bucket lowers the false negative rate, which is the likelihood that a relevant result is missed by the search. 'numHashTable' is the number of hash tables to be used. Each hash table uses a different random projection, so increasing the number of hash tables leads to increasing the probability of finding the nearest neighbors, but at the cost of increased computational requirements. Different ranges of values for both key parameters were experimented. We tested 'bucketLength' values from 2 to 4 and 'numHashTable' values from 3 to 7. Based on the hyperparameter tuning using the small and full validation set, the following parameters were chosen:

buckeLength	**numHashTable**
4	3

Figure 10. Best Parameters Chosen

Using the target hyperparameters, the Nearest Neighbor model was trained using the full training data and was evaluated using both the validation set and the test set.

	Validation	Test
Small	0.006658	0.013976
Full	0.005358	0.012000

Figure 11. MAP@100 for Nearest Neighbor Search

From the results, MAP@100 is higher in small validation and test set than the score for full validation and test set. One potential factor that could have affected the lower performance on the full dataset is size and diversity of the datasets. Smaller datasets might not capture the full diversity of user-item interactions and might be biased towards certain types of interactions, leading to higher performance, but lower performance when the model is applied to the full, more diverse dataset.

References

[1] Kumar, A. (2022, April 26). Hold-out method for training machine learning models. *Data Analytics*. https://vitalflux.com/hold-out-method-for-training-machine-learning-model/
[2] Evaluation metrics—Rdd-based api—Spark 3. 0. 1 documentation. (n.d.). Retrieved April 28, 2023, from https://spark.apache.org/docs/3.0.1/mllib-evaluation-metrics.html#ranking-systems
[3] ChatGPT, identifying and correcting errors. Used from April 12, 2023 to May 16, 2023. Retrieved from https://openai.com/chatgpt/
[4] *Locality Sensitive Hashing*. https://george-jen.gitbook.io/data-science-and-apache-spark/locality-sensitive-hashing. Accessed 16 May 2023.