Final Project Checkpoint Report - Group 6

Fangru Li
CDS
New York University
New York, NY
fl1099@nyu.edu

Robin Wang
CDS
New York University
New York, NY
jw5487@nyu.edu

Zejun Zhang
CDS
New York University
New York, NY
zz4140@nyu.edu

Introduction

The objective of this project is to develop and assess a song recommendation system based on collaborative filtering using the Alternative Least Squares (ALS) model in Apache Spark, implemented with Python. The system utilizes the interactive user records and their listened recording from ListenBrainz Dataset, which are stored on the Hadoop Distributed File System (HDFS). To evaluate its performance, the ALS model is compared to a single-machine implementation of LightFM in terms of efficiency and accuracy. Also, we implemented the Annoy model to accelerate searches.

Data Overview and Processing

Data Overview: The main dataset we used was extracted from ListenBrainz takes in the form of implicit feedback, with each row composed of a user ID, a recording ms ID, and a timestamp. This dataset was pre-split into two set, the train set for model training and test set for model evaluation.

Data Processing - Data Down-sampling: Due to the sheer size of the dataset, a downsampling method on the training set was considered in regard to the computational cost in time and memory. First, we decided to subsample the data set by 50% to ensure we have a simplified data set while keep a low RMSE difference between subsampled data set and full set low. Since we want to split the training data into a smaller set of training data and validation data for parameters tunning, we removed some data that we believe not effective to the our recommender system modeling. Specifically, we removed all the users who had less than 50 records. Then, each users' records were splitted into training data set and validation data set with an 80/20 ratio. Until now, both training data set and validation data set have three rows (userid, recordingid, timestamp), and considering the timestamp is not related to our ALS model a lot, this column was dropped. After that we count how much time each user listens to each recording, and add the counting number as "count" column into both data set.

Data Feature Transformation: To meet the requirement of ALS model, all columns of our data set should be in integer type. Thus, we tried StringIndex method, but due to the limitation of executors and memory we were assigned from Dataproc, this method never success. In this project, we used another method to implementation the same result. First, we select the distinct recordingid from the original training set, and assign a unique integer value to each recordingid. Then, applying an inner join to the training set and validation set separately, so that we can achieve the same result as the StringIndex method without any "killed" or "Error 143" issue. At the end of this step, we also repartitioned the train set and test set by 5000 to improve the execution plan.

Popularity Baseline Models

In order to build a baseline to our recommendation model, we start with the most basic method: uniformly recommending the most popular tracks. It is worth mentioning that the most popular tracks are considered as 500 recordings that have the most number of distinct users who have played. To evaluate our baseline model, we mainly use Mean Average Precision (MAP), which takes into account both the precision and the rank of each relevant document in the retrieved list. However, the average MAP value is around 4.63×10^{-5}.

Based on settings above, we next pursue hyperparameter tuning. To make the process more streamlined, we make use of the ALS.fitMultiple() which enables us to train and tune multiple hyperparameter combinations efficiently. We primarily utilize 4 hyperparameters: 'rank', 'maxIter', 'regParam' and 'alpha'. In particular, 'rank' controls the number of latent factors used to represent users and items. 'maxIter' defines the maximum number of iterations to run during training. 'regParam' is a regularization term to avoid overfitting. Additionally, 'alpha' is used to balance the weight between user- and item-specific factors.

ALS Model

ALS Model and Our Goal: ALS (Alternating Least Squares) model in Apache Spark represents a powerful tool for collaborative filtering recommendation systems. The ALS model is a matrix factorization-based recommendation algorithm that captures the underlying relationships between users and items by decomposing the user-item interaction matrix. This model is commonly used to provide personalized recommendations to users.

In this project, given the dataset, we utilize the ALS model in Spark to implement a music recommendation system aimed at recommending 100 songs that are suitable for users based on their listening counts. By leveraging the ALS model and the powerful computing capabilities of Spark, we can build an efficient and accurate music recommendation system that provides a personalized music experience for each user.

Evaluation Values: Based on our goals, which was to prioritize songs with higher confidence levels in the top positions while recommending music that is suitable for users, we chose three metrics to evaluate our recommender system performance. To aim to accurately recommend items of interest to users within the given recommendation list, we applied **Precision at K**, which is a metric that measures how many recommendations within the top k are accurate. Then, **NDCG**（**Normalized Discounted Cumulative Gain**）is a metric used to evaluate the quality of ranking in recommendation systems. NDCG values typically range from 0 to 1, when the ranking of the recommendation results aligns closely with the relevance of the items, NDCG achieves higher scores. To evaluate the comprehensive quality of our recommender systems, we add another matric **Mean Average Precision(MAP)**. This matric provides a more comprehensive measurement of the performance of a recommendation system, assigning higher weight to recommendations that are both accurate and ranked higher.

Parameter Tuning: In this part, we tuned three parameters, which include rank, regularization parameter(ref, 1), and alpha to improve the ALS model performance on validation set. Considering the limitation of computing time and memories, we selected three values for each parameter. We selected [10, 30, 50, 100] for **rank**, [0.005, 0.05, 0.1] for **reg_params**, and [1, 20, 50] for **alpha**. Firstly, considering the independence of rank to the other two parameters, we set fixed regularized parameter and alpha, and as the following figures show, the evaluation values monotonically increase with the increased rank. However, larger rank leads to longer time to train the model and overfitting problem. Thus, although we rank = 100 gains higher evaluation values, we use **rank = 50** to save time and memories.

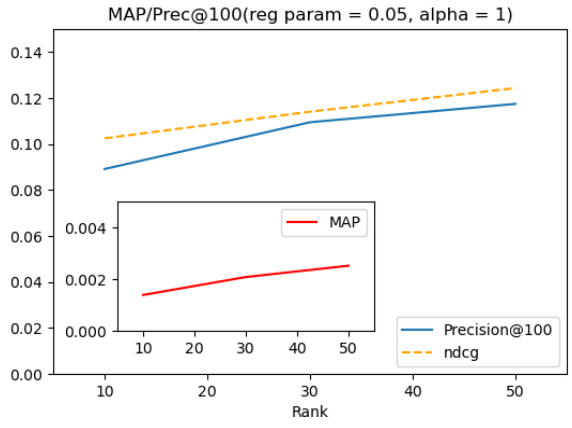

Figure 1: ALS Model Performance w.r.t. Rank

Next, for the three values we picked for regularization parameters and alphas, we did grid search to find the best combination:

rank = 50, reg param = 0.005			
alpha	1	0.1	0.01
MAP	0.002508	0.002351	0.001975
Prec@100	0.119783	0.050063	0.045614
ndcg	0.126354	0.056141	0.051157

rank = 50, reg param = 0.05			
alpha	1	0.1	0.01
MAP	0.003507	0.002228	0.001576
Prec@100	0.061031	0.049599	0.038453
ndcg	0.067295	0.054678	0.052671

rank = 50, reg param = 0.1			
alpha	1	0.1	0.01
MAP	0.003419	0.002016	0.001265
Prec@100	0.0598	0.047964	0.031845
ndcg	0.065479	0.052469	0.050039

Table 1: Grid Search Form Based on Different reg_params and alphas

With the parameter tuning, we finally picked the best combination: **rank = 50, reg_param = 0.05, and alpha = 1**. The ALS model with these parameters achieved:

MAP	0.0025167
Precision@100	0.117459
ndcg	0.12434666

Table 2: Evaluation on the Test Data

Extension I: LightFM

Introduction. This portion of the report presents a detailed analysis of a music recommendation system based on the LightFM library.

Model Training and Evaluation. The LightFM model is trained and evaluated using the Weighted Approximate-Rank Pairwise (WARP) loss function. A grid search approach is employed to tune the model's hyperparameters, which include the rank (no_components) and regularization (user_alpha). (ref 2) The precision_at_k metric is used to evaluate the model's performance on the validation set. The best model's parameters are determined by the highest precision_at_k score.

Parameter Search and Timing Analysis. The following parameter combinations are tested in the grid search: Ranks: 20, 30, 40, 50; Regularizations: 0.005, 0.01, 0.1. For each parameter combination, the model is trained, and the precision at k=100 is calculated. Additionally, the time spent on training and evaluating each model is recorded.

Results. The best model's parameters are identified as rank=20 and regularization=0.005. The final model is then trained using these parameters and evaluated on the validation set. The time spent on training and evaluating the best model is also recorded. (Please see the heat map below)

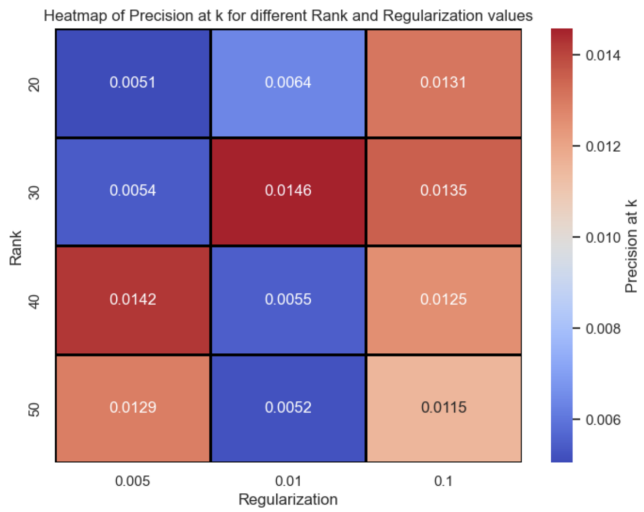

Figure 2: Heatmap of Precision at k for Rank and Reg. Values

The best-performing model has rank=30 and regularization=0.01, with a precision at k=100 of 0.0146 and a training time of 1100.234. This combination might be the best due to the following reasons: First, the choice of rank plays a significant role in the quality of the model. The rank determines the number of latent factors used in the matrix factorization process, which can affect the model's ability to capture the underlying structure in the user-item interaction data. A rank of 30 might provide an optimal balance between model complexity and the ability to capture meaningful patterns in the data. A lower rank, such as 20, may not capture enough information to make accurate recommendations, while a higher rank, such as 40 or 50, could introduce overfitting due to excessive complexity. Second, the regularization parameter influences the model's ability to generalize to new data by controlling the model's complexity. A regularization value of 0.01 appears to be a suitable choice for this particular dataset. Lower regularization values (e.g., 0.005) might allow the model to fit the training data too closely, making it prone to overfitting and reducing its performance on the test set. In contrast, a higher regularization value (e.g., 0.1) could result in underfitting by penalizing the model's complexity too much, leading to a loss in predictive accuracy. Lastly, the dataset's unique characteristics may contribute to the observed performance of the model with rank 30 and regularization 0.01. The interaction data could have a specific structure that is optimally captured by this particular configuration. This structure may not be as well-represented in models with different ranks or regularization values.

Results comparison (ALS and LightFM)

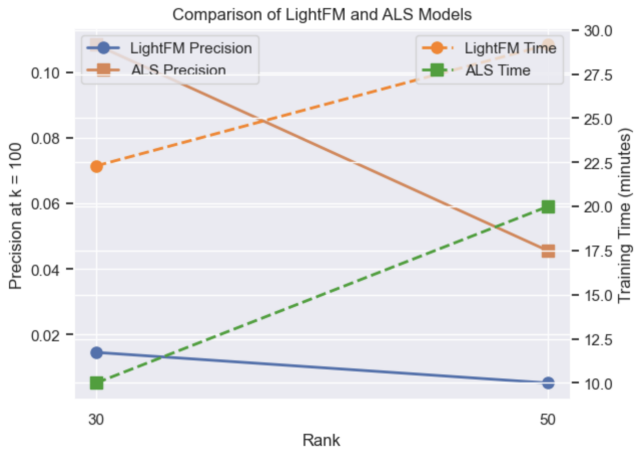

Figure 3: Comparison of LightFM and ALS Models

At rank 30, LightFM achieves a precision at k = 100 of 0.014569, while ALS achieves a higher precision of 0.108474. However, LightFM takes approximately 22.3 minutes to train, while ALS only takes around 10 minutes.At rank 50, LightFM's precision at k = 100 decreases to 0.005235, whereas ALS still outperforms LightFM with a precision of 0.04561351069. For the model fitting time, LightFM takes about 29.2 minutes, while ALS takes 20 minutes to train.

Based on our comparison, it is evident that ALS outperforms LightFM in terms of precision at k = 100 for both rank values. Moreover, ALS demonstrates a significantly lower model fitting time, making it a more efficient model for this specific task. One of the reasons behind ALS's superior performance could be its ability to handle larger data sets and cope with data sparsity issues more effectively compared to LightFM. ALS can leverage matrix factorization techniques to capture latent factors, resulting in better recommendations, even in the presence of the cold-start problem. On the other hand, LightFM, a hybrid model, might require more time to learn the item and user features, which could lead to a longer fitting time and lower precision at higher ranks.

Conclusion: In summary, the best model's parameters (rank=30 and regularization=0.01) provide a balance between expressiveness and generalization, resulting in improved performance. The increased training time is a trade-off that should be considered when deploying the model in a real-world situation.

Extension II: Fast Search with Annoy Algorithm

Although brute force search is a straightforward approach to find the nearest neighbors in a dataset, it can be time-consuming to exhaustively compute the dot product between the user factor and each item factor, especially in high-dimensional spaces. To address this problem, we utilize the Approximate Nearest Neighbor (Annoy) library in our recommender system, which could accelerate the retrieval process required for making recommendation to users.

Annoy algorithm consists of a set of randomized binary decision trees, in which each tree represents an index that enables fast retrieval of approximate nearest neighbors. In particular, a random vector from the dataset was selected as the root node of the tree, and then, the vector is recursively split into child nodes based on random hyperplanes. This process creates the index structure and ends up until the stopping criterion is satisfied. When a query vector is provided, the Annoy algorithm traverses down the search trees, and compares the query vector with hyperplanes at each node. Based on the side of the hyperplane the query vector falls, it proceeds to the corresponding child node. This procedure continues until the query vector landing on the terminal node, which contains vectors that are approximate nearest neighbors of the query vector.

To implement Annoy in this case, we need to make use of the parameters (rank = 50, reg params = 0.05, alpha =1) derived from the optimal ALS model we obtained above, and export the representations from HDFS to our local machine. Then, we build a forest of trees based on all learned track factors with Annoy's python API. When we input a user factor in the querying phrase, Annoy traverses the entire forest of trees and identifies the set of neighboring items for the given user factor, which could be considered as the candidate set. It then calculates the dot product between the user factors and each item factor within the candidate set. At the final step, Annoy would generate 100 recommendations based on the sorted dot product value.

We mainly compare the computational time as well as the accuracy (MAP and Precision@100) between brute force search and the Annoy algorithm with varied numbers of trees on the validation set. The results are shown in the table below. As we expected, Annoy dramatically reduces the evaluation time at the expense of certain accuracy. However, it is worth noting that the precision increases as the number of trees grows, and the difference between Annoy and brute force search accuracy shrinks accordingly.

# of trees	MAP	Prec@100	Evaluation Time (s)
50	0.001892	0.038721	635.47
100	0.002479	0.041894	1074.58
200	0.002935	0.051986	1298.62
Brute Force	0.003507	0.061031	2134.67

Table 3: Brute Force Search vs. Annoy

In conclusion, Annoy provides approximate nearest neighbors rather than exact ones, trading off some accuracy for improved efficiency. With Annoy, recommender systems can efficiently handle large-scale datasets and high-dimensional feature spaces commonly found in real-life scenarios.

References:

1.Collaborative Filtering: Scaling of the regularization parameter
https://spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html

2. LightFM. "LightFM: A Python implementation of a hybrid recommendation algorithm." GitHub, https://github.com/lyst/lightfm.

Link to the Repository:

https://github.com/nyu-big-data/final-project-group-6

Contributions:

Member name	Contribution
Fangru Li	Parameter Tuning, Extension II, Final Report
Robin Wang	Data Processing, Extension I, Final Report
Zejun Zhang	Data Processing, Parameter Tuning, ALS Model, Final Report