Recommender System

Spring 2023 DS-GA 1004 Final Project

https://github.com/nyu-big-data/final-project-group-176.git

Sabrina Zhu
Center for Data Science
New York University
New York, NY USA
cz2767@nyu.edu

Philip Xing
Center for Data Science
New York University
New York, NY USA
hx648@nyu.edu

Edward Chen
Center for Data Science
New York University
New York, NY USA
yc6292@nyu.edu

1. Introduction

In the realm of digital music streaming, providing personalized song recommendations is a critical component of user engagement and satisfaction. For our final project in the Spring 2023 DS-GA 1004 course, our group decided to delve into the fascinating world of recommender systems, specifically focusing on music recommendation. Our project is a comparative study of multiple recommendation algorithms, assessing their performance and efficiency in terms of predictive accuracy and computational time. The algorithms we've explored include the Popularity-Based Model, the Alternating Least Squares (ALS) Model, LightFM, and a Fast Search approach using the ANNOY library. Through this project, we aim to understand the strengths, weaknesses, and appropriate use-cases for each model, with the ultimate goal of improving music recommendation quality and efficiency.

MAP@K is a ranking metric that considers both the relevance and the ranking of recommended items. It is chosen because it aligns with the goal of prioritizing relevant recommendations, handles sparse and imbalanced data, accommodates varying user behavior, and allows for easy model comparison and enhancement evaluation. Thus, we chose this metric to evaluate our models.

2. Data Preprocessing

Our data preprocessing step was implemented using PySpark due to the large size of our dataset. The dataset was obtained from the ListenBrainz dataset, which consists of implicit feedback from music listening behavior, spanning over 6,800 distinct users and more than 50 million song interactions.

The preprocessing included data filtering, feature extraction, and partitioning of the data into training, validation, and test sets.

Data Loading: We loaded the training data from 'interactions_train_small.parquet' and the test data from 'interactions_test.parquet'. These files contained interactions between users and tracks. Due to the scarcity of computing resources and the long wait times to perform recommendations on even the small dataset, we were not able to implement our model on the entire dataset.

Filtering: Given the implicit nature of the feedback, we decided to filter out users who had less than 50 interactions with tracks to ensure the robustness of our recommender system. This was done to mitigate the potential impact of the cold start problem caused by users with sparse interaction data on the performance of our recommender system.

Track ID: Each song in the dataset is identified by a unique string id called 'recording_msid'. However, the ALS model requires numeric ids. Thus, we assigned a unique 'track_id' to each unique 'recording_msid'.

Data Partitioning: After filtering and assigning track id, we split our dataset into training and validation. The splitting process involved assigning a unique 'interaction_id' to each interaction in the training data. We then performed a stratified sampling by 'user_id' to split the filtered training data into a training set (80% of interactions) and a validation set (20% of interactions). The stratified sampling process mitigated the cold start problem.

Data Saving: The preprocessed training, validation, and test sets were saved back to parquet files for future use. They were grouped by 'user_id' and ‘track_id’ to count the number of times a particular user listened to a specific track. This was done to reduce the size of the dataset to enable efficient retrieval. Additionally, validation and test data were further reduced to ‘user_id’ and a set of track_ids they have interacted with to prepare the data for model performance evaluation. This was done to further reduce data saving and reading times.

3. Popularity Based Model

3.1. Baseline Implementation

We implemented a baseline popularity model, which recommends the most popular songs to all users. It takes into account the number of unique listeners as the most crucial factor. The popularity measure is calculated as the ratio of distinct listeners for track i over the total number of plays for track i plus a hyperparameter beta.

3.2. Baseline Performance

The baseline popularity model performance for different values of β on the training and validation sets is presented in the following table:

Beta	Training MAP	Validation MAP
0	0	2.42e-08
1000	1.93e-04	1.86e-04
10000	8.96e-04	8.96e-04
100000	9.67e-04	9.63e-04
200000	9.79e-04	9.85e-04
500000	9.77e-04	9.79e-04
1000000	9.78e-04	9.80e-04

Based on the results, the best value of β is 200,000, which yields a validation MAP@K of 0.000985.

The popularity baseline MAP@K at 100 on the test set is 0.000982.

4. ALS Model

In this project, we utilized the Alternating Least Squares (ALS) model, a matrix factorization algorithm that's well-suited for large-scale recommender systems. The ALS model identifies latent factors, or hidden features, from implicit user feedback data to generate high-quality music recommendations. A key part of our process was tuning the model's hyperparameters, such as rank, regularization parameter, and alpha, to achieve optimal performance. This section delves into our ALS model implementation, the hyperparameter tuning process, and the ensuing results.

4.1. ALS Implementation

Following the data preprocessing, we implemented the Alternating Least Squares (ALS) model using PySpark's ML library. Given the large size of our dataset and the inherent parallelization capabilities of Spark, the choice of ALS was appropriate.

The ALS model was constructed using the implicit preferences variant, which is more suitable for our dataset as it contains implicit feedback from users' music listening behavior. It was initialized with several hyperparameters:

1. Rank: This parameter represents the number of latent factors in the model. Essentially, it describes the quantity of hidden elements in the vectors representing users and items. The complexity of the model is determined by the rank; a larger rank may help the model to recognize complex patterns, but it may also lead to overfitting and increased computational expenses. We initially hoped to train and validate our model with ranks of 10, 50 and 100. Although, even when rank was set to 10, given the scarcity of computational resources, the long wait time (~ 30 min) to train and evaluate and the frequent crashes of the HPC environment, we were forced to adapt this plan. We found the optimal lambda and alpha combination with rank equal to 10. We then trained our model with this optimal pair on rank set to 50.

2. Regularization parameter (lambda): This parameter helps to deter overfitting by penalizing large parameters. A larger value implies stronger regularization. We tested the model with values 0.01, 0.1, and 0.5, since performance seems to peek in this range given our early test runs.

3. Alpha: This parameter is unique to the ALS model's implicit feedback variant. It governs the base confidence in observations of preference or non-preference. We used alpha values of 0.1, 0.5, and 1, since performance seems to peek in this range given our early test runs.

The ALS model was then fitted on our training data, which was represented by a Spark DataFrame containing three columns: 'user_id', 'track_id', and 'count'. The 'user_id' and 'track_id' columns represented the user-item pairs, while the 'count' column represented the implicit feedback or interaction frequency between each user-item pair.

The fitted model was used to generate predictions for the validation set. These predictions were evaluated using the Mean Average Precision at K (MAP@K) metric.

The process of fitting the ALS model and evaluating its predictions was repeated for each combination of hyperparameters. The combination that resulted in the highest MAP@K on the validation data was chosen as the best hyperparameters.

4.2. ALS Performance

In this project, we've used a latent factor model, more specifically, the Alternating Least Squares (ALS) model.

Here is a summary of our results:

Rank	Lambda	Alpha	MAP
10	0.01	0.1	0.020060
10	0.01	0.5	0.024306
10	0.01	1	0.011902
10	0.1	0.1	0.010205
10	0.1	0.5	0.030774
10	0.1	1	0.028814
10	0.5	0.1	0.009145
10	0.5	0.5	0.022576
10	0.5	1	0.013002
50	0.1	0.5	0.041053
50	0.1	0.5	0.032762

We fitted the model on the training data for each hyperparameter combination, made recommendations for a user subset, and evaluated these recommendations using Mean Average Precision at K (MAP@K). The best parameters were those that produced the highest validation data MAP.

The evaluation of our latent factor model relied on Mean Average Precision at K (MAP@K), a ranking metric that assesses the average precision of recommended items up to a certain cut-off point, K, we set it to 100.

During the hyperparameter tuning phase, the optimal parameters were determined to be a rank of 10, lambda of 0.1, and alpha of 0.5, yielding a validation MAP@K of 0.030774. However, due to computational limitations, we had to limit the rank to 10. When we trained the model with a higher rank of 50, using the optimal lambda and alpha, the validation MAP@K increased to 0.041053.

In the test phase, we used the model with a rank of 50, lambda of 0.1, and alpha of 0.5. The resulting test MAP@K was 0.032762. This result suggests that a model with these hyperparameters can produce relatively high-quality recommendations and shows a great improvement over our baseline model. Despite the challenges posed by computational constraints, the model still managed to perform well, demonstrating the value and efficiency of the ALS model for making recommendations.

5. Extension 1: LightFM

In this project, we explored the LightFM model, a Python-based hybrid matrix factorization algorithm suitable for music recommendation. It handles implicit feedback effectively and was implemented alongside the ALS model. We created an interaction matrix, trained LightFM using the 'warp' loss function, and evaluated its performance using MAP@K for consistency. Here's an overview of the LightFM implementation, performance metrics, and a comparative analysis with Spark ALS.

5.1. LightFM Implementation

We replicated our PySpark preprocessing steps using Python Pandas, and translated user and item IDs into a contiguous range of integer IDs to fit LightFM's requirements.

After transforming the training data into a sparse matrix, we trained a LightFM model using the 'warp' loss function, suitable for implicit feedback data, over 30 epochs with 16 threads.

Only users present in the training set were included in the test set. We computed model scores for each user-item pair, selecting the top 100 items. Using the intersection of these items and the true labels, we computed the MAP@K.

The subsequent sections will evaluate the LightFM model's performance and efficiency and compare it to Spark's ALS model.

5.2. LightFM Performance

Given the size of the dataset and the slowness of the single machine implementation, we weren't able to implement

LightFM on the full dataset. We were forced to train and evaluate only using 1% of the data. This suggests the limitation of the single machine LightFM model compared to the Spark ALS model

The LightFM and Spark ALS models were evaluated based on computational efficiency and MAP@K score.

LightFM took 155.62 seconds to train even on 1% of the training data, demonstrating its scalability issue and inability to handle large datasets like ALS.

On the reduced dataset, Spark ALS took 10.07 seconds to train, which is much faster than the LightGM. LightFM shows poor computational efficiency.

The MAP@K score for LightFM was 0.001214, indicating poor predictive performance.

Training on this reduced data set, Spark ALS achieved a MAP of 0.0032762. This result suggests that LightFM underperforms Spark ALS.

In summary, LightFM struggles with scalability and predictive performance for larger datasets. Spark ALS offers better scalability, computational efficiency and predictive performance when handling full datasets.

6. Extension 2: Fast Search with ANNOY

In the fast search extension of our project, we utilized a spatial data structure to accelerate the search process at query time and improve our basic ALS models. We employed the Annoy (Approximate Nearest Neighbors Oh Yeah) library developed by Spotify for this purpose. Our choice to use Annoy was guided by its efficiency and speed, especially for large datasets. This report details the implementation of this extension and evaluates the efficiency gains and changes in accuracy induced by the approximate search.

6.1. Fast Search with ANNOY Implementation

1. Preprocessing: Before implementing the fast search, we preprocessed the data to make it suitable for the task. Instead of dropping users with less than 50 interactions in our basic ALS model, we filtered the top 500 users here to maintain a manageable dataset size.
2. Implementation: The implementation is an upgraded version of the basic ALS model. After training the basic ALS model, we used Annoy to create an index from the item factors generated by the basic model. This step was instrumental in transforming our ALS model parameters into a format that could be efficiently queried using Annoy.

We then used the Annoy index to generate recommendations for each user. Annoy's efficient search algorithm allowed us to quickly retrieve the nearest items for each user.

6.2. Fast Search with ANNOY Performance

1. Computational Efficiency: The total time taken to create the Annoy index was approximately 0.48 seconds, a remarkably short period demonstrating the computational efficiency of Annoy in indexing the dataset. The recommendation time was also efficient, taking only about 0.14 seconds.

2. Predictive Performance: The MAP@K for Annoy is 0.048, which is significantly higher than the Spark ALS model alone (MAP@K of 0.0065 on the reduced dataset). This suggests that the Annoy-based Fast Search method provides superior performance in terms of accurately ranking items that users are likely to interact with.

In summary, the Fast Search approach with Annoy has demonstrated high computational and evaluation efficiency and superior predictive performance. This makes it a compelling choice for scenarios requiring quick recommendation generation and high-quality results. However, it's also important to consider the nature and size of the dataset, as different approaches may yield different results based on these factors.

7. Conclusion

In conclusion, our exploration of various recommender systems offered valuable insights into their respective strengths and limitations. The Popularity-Based Model served as a useful baseline, while the ALS model demonstrated remarkable performance in dealing with large-scale datasets. The LightFM model, although worked with smaller datasets, faced challenges with scalability and predictive performance. Finally, the Fast Search approach with ANNOY displayed superior computational efficiency and predictive performance, suggesting it to be a compelling choice for scenarios requiring quick recommendation generation and high-quality results.

8. Contribution

Sabrina Zhu: ALS and Extension 2
Edward Chen: ALS and Extension 1
Philip Xing: Extension 1 and Report