Collaborative-Filter Recommender System

DSGA-1004 Final Project Report

Drishti Singh
New York University
NY USA
ds6730@nyu.edu

Pranjal Srivastava
New York University
NY USA
ps4379@nyu.edu

Vaishnavi Rajput
New York University
NY USA
vr2229@nyu.edu

ABSTRACT

In this project, our goal is to develop a recommender system using collaborative filtering on the ListenBrainz dataset. Recommender systems are specialized information filtering systems designed to predict user preferences. The two primary types of recommendation systems are collaborative filtering and content-based filtering. Collaborative filtering analyzes user behavior and preferences to identify similarities and patterns among users. By finding users with similar tastes and preferences, the system can recommend items that have been liked or highly rated by those similar users. On the other hand, content-based filtering focuses on the attributes and characteristics of items themselves. It examines the features or metadata associated with items and recommends similar items based on a user's past interactions. By understanding the content of the items, the system can suggest relevant items that align with a user's preferences.

The code for the project can be found at the following GitHub link: https://github.com/nyu-big-data/final-project-group-29.

1 INTRODUCTION

With the growing popularity of e-commerce businesses, recommender systems help overcome the challenges customers face when navigating through a wide range of products by suggesting personalized or best-suited options. These systems rely on various types of input, with the most convenient being high-quality explicit feedback. Users directly provide this feedback by expressing their interests in products or services. For example, platforms like Netflix and IMDB collect star ratings for movies. However, explicit feedback is not always readily available. In such cases, recommender systems utilize implicit feedback, which is more abundant and reflects user opinions indirectly through their observed behavior. Implicit feedback includes any user interactions data such as purchase history, browsing history or search patterns. For instance, if a user has purchased several books from the same author, it can be inferred that the user likely enjoys the work of that specific author.

Therefore, a collaborative filtering approach is taken to build our recommender system, relying on users' past behaviors without explicitly creating profiles. To achieve this, we used ListenBrainz, a crowd-sourced open to public music listening dataset that tracks listen history of users across different platforms. We will utilize two datasets from ListenBrainz, namely "interactions" and "tracks," which have already been partitioned into test and train splits. The train split contains data from 2015 to 2018, while the test split contains data from 2019.

2 METHODOLOGY

2.1 PRE-PROCESSING

Initially, we decided to utilize interaction data for the project's initial checkpoint. This data consisted of three columns: user_id, recording_msid, and timestamp. As the project progressed, it became apparent that we needed to incorporate tracks data with interaction data to identify identical tracks with the same recording MBID. This step aimed to reduce the data further down by identifying identical tracks.

To streamline the data for further analysis and ensure cleanliness and organization, we employed various pre-processing techniques. After joining interaction and tracks table, two separate dataframes were created: one with MBID and one without. In the MBID-present dataframe, we employed translations to group and retrieve all recording MSIDs associated with the same MBID. Later, we merged both dataframes and eliminated unnecessary columns to improve data cleanliness.

In the merged dataframe, we calculated the number of listens for each recording MSID by each user. This information was necessary for calculating our sparse matrix for one of the alternate approach we experimented with. Additionally, to avoid overfitting caused by less user interactions, we computed the mean and median of our dataframe. We then filtered out rows with listens below the median plus three threshold and normalized the data to center it around zero, while also removing records any negative mean values. We chose adding 3 in our threshold as buffer as other values were significantly affecting total records in the dataframe.

To make the data compatible with the latent factor model, we created a list of all unique recording MSIDs from both small and complete training and test dataframes. We assigned categorical codes to each recording MSID and constructed a new dataframe with columns recording_msid and codes. We then performed a left join operation, merging this new dataframe with the existing train and test dataframes based on the recording MSID. At this point, all the necessary preprocessing was completed.

Subsequently, we divided the training dataset further into training split and validation split in a 70:30 ratio. The training split included 60% of the validation data to ensure that all users were accounted for during the model training.

2.2 BASELINE POPULARITY MODEL AND ITS EVALUATION

In the next phase of the analysis, both the small and complete sizes of the training split were utilized to develop and evaluate the baseline popularity model that recommends popular songs based on user interactions. The model was developed using two distinct approaches:

(i) Approach 1: Calculate the average number of listens for each track per user.

(ii) Approach 2: Calculate the raw count of listens per track on the entire dataset.

To test the performance of the baseline popularity model, ranking technique was used on the validation and test data splits to determine the top 100 songs listened by each user which acted as ground truth. These top 100 songs were compared to the prediction of the baseline model and accuracy was evaluated using RankingMetrics. Precisely, Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG) were used for the calculation and the following results were obtained:

Dataset	MAP@100	NDCG@100
Small Validation	4.901960784313726e-07	8.02964105594774e-06
Validation	5.3670527366601884e-08	2.5602350374031623e-06
Test	1.5528946403147546e-05	5.832928007163257e-05

Table 1: Results of evaluation on popularity baseline model with average listen (Approach 1)

Dataset	MAP@100	NDCG@100
Small Validation	0.0002762836633979165	0.002881124712965931
Validation	0.00045677154598667386	0.0043396770058887835
Test	9.285713485266364e-05	0.0012220825266727297

Table 2: Results of evaluation on popularity baseline model with total listen (Approach 2)

Table 1 and Table 2 show the results of the evaluation using both the approaches. The accuracy of both the metrics were considerably low for approach 1 in comparison to approach 2.

2.3 ALTERNATING LEAST SQUARES ALGORITHM

In this project, we employed the Alternating Least Squares (ALS) algorithm, a widely utilized method for matrix factorization. ALS facilitated the decomposition of the user-item interaction matrix into latent feature matrices, enabling the extraction of valuable insights. Our objective was to learn latent factors that captured both user preferences and item characteristics. This iterative process involved optimizing the factor matrices to minimize the reconstruction error, ensuring a closer alignment between predicted and observed ratings. By harnessing the acquired latent factors, we generated personalized music recommendations for users, drawing from their historical listening patterns and similarities with other users. The collaborative filtering approach implemented in our system leveraged the collective wisdom of the user community, leading to accurate recommendations.

3 EVALUATION

3.1 HYPER-PARAMETER TUNING

During the training process, our model primarily focused on tuning three hyper-parameters of the ALS algorithm: Rank, Regularization Penalty, and Implicit Feedback parameters. Additionally, we also tried to use other hyper-parameters like implicitPrefs and coldStartStrategy to optimize the model's performance and achieve the highest possible score on unseen data, specifically the test set. In addition, we also experimented changing numUsersBlocks and repartitions for better parallelism.

3.2 RMSE (ROOT MEAN SQUARED ERROR)

We evaluated the model using the root mean square error metric. It essentially measures the deviation of our model's prediction from the observed values. The expression in Figure 1 describes the mathematical formula used for calculating RMSE.

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

Figure 1: RMSE definition

3.3 PRECISION AT K

Precision at k is a performance metric used to evaluate the accuracy of a recommendation system. It measures the proportion of relevant items among the top-k recommendations provided to a user. The expression in Figure 2 describes the mathematical formula used for P@K.

$$P@K = \frac{\sum \text{Relevant items in top K recommendations}}{\sum \text{Items in top K recommendations}}$$

Figure 2: Precision at K definition

3.4 AUC

This score indicates the frequency with which a positive interaction is ranked higher than a negative interaction.

4 EXTENSIONS

4.1 LIGHTFM

We extended a baseline collaborative filtering recommender system through the implementation of two enhancements. One of the extensions involved the use of the LightFM library, a Python-based hybrid recommendation system. The following steps were followed for our extension:

1. Data processing: The data was processed to fit the LightFM model's requirements. The unique identifiers were generated for each user and item of the dataset.

2. Model Construction: The LightFM model incorporates the WARP (Weighted Approximate-Rank Pairwise) loss function, which is particularly well-suited for hybrid models and scenarios involving implicit feedback. In this study, a total of 30 components were utilized in the model construction. To optimize the model's performance, a learning rate of 0.01 was selected. These choices were made with the aim of enhancing the model's efficacy and effectiveness in capturing user preferences in a hybrid recommendation system context.

3. Model Training: The training process of the model involved utilizing interaction data in the form of a sparse matrix. This choice of data structure proved to be highly efficient for handling datasets with a large number of zero entries, optimizing computational resources and overall performance.

4. Evaluation: The evaluation of the model's performance utilized the precision_at_k metric, which quantifies the accuracy of recommendations within the initial 'k' predictions. To ensure a fair comparison, precision was assessed separately on both the training and validation sets. The validation set was carefully filtered to include only users and items that were also present in the training set, ensuring a consistent evaluation framework.

5. Model Interpretation: The precision_at_k results provided an objective measure of the model's performance. Further evaluation involved comparing results to the baseline model's performance and inspecting the actual recommendations made by the model, along with applying additional evaluation metrics.

4.2 FAST SEARCH

The goal of Fast Search is to enhance the efficiency of recommendation retrieval by utilizing the Annoy library, which provides an approximate nearest-neighbor search algorithm. The approach involves loading the pre-trained ALS model, extracting item and user latent factors, creating an Annoy index, and generating recommendations based on nearest neighbors. The performance of the model is evaluated using metrics such as accuracy and AUC on validation data. Additionally, query time is measured to assess the efficiency of the recommendation retrieval process.

To implement Fast Search, we began by loading the ALS model and extracting the latent factors for both items and users. Next, we indexed these factors to enhance the speed of retrieval. We then performed evaluation on the validation dataset. By creating an Annoy index with the specified number of trees, we established the foundation for efficient recommendation retrieval. This index was populated with the item latent factors. Utilizing the Annoy index, we generated recommendations for users based on their nearest neighbors. To assess the performance of the recommendations, we calculated metrics such as accuracy and AUC. Additionally, we measured the query time to evaluate the overall efficiency of the recommendation retrieval process.

5 RESULTS

Table 3 shows the results of RMSE evaluator on small validation set for various hyper-parameters in our first approach where we didn’t apply data normalization for small dataset.

Table 4 and 5 shows the results of RMSE evaluator on complete test set for various hyper-parameters in our second approach with data normalization for small and complete dataset.

MaxIter	Rank	Reg = 0.1
20	20	6.25
	30	12.39
	40	12.38

Table 3: RMSE values for different hyper-parameters on small dataset for first approach

MaxIter	Rank	Reg = 0.01	Reg = 0.1
20	20	49.89	49.89
	30	49.87	49.86

Table 4: RMSE values for different hyper-parameters on small dataset for second approach

MaxIter	Rank	Reg = 0.01	Reg = 0.1	Reg = 1
10	20	10.01	10.03	10.03
	30	10.02	10.02	10.03
20	20	9.91	9.95	9.95
	30	9.92	9.96	9.95

Table 5: RMSE values for different hyper-parameters on complete dataset for second approach

For LightFM, we hyper-tuned the parameters number of components = 30 and alpha = 0.01. However, the script failed in the middle of the execution due to space issue.

6 CHALLENGES

Throughout the project, we faced several challenges, and the most significant one was related to resource allocation on Spark. While we understand that HPC is a shared resource, the limited availability of resources had a considerable impact on our progress, and we had to extend our self-imposed deadlines.

Dealing with big data is a complex task, and we encountered numerous memory errors, which prompted us to conduct additional research and optimize our code further. We were not able to test out tuning many hyper-parameter combinations as we imagined due to Java Heap space errors or resource allocation issues.

We also noticed as iterations increased, the jobs were killed, or Java heap space error would come up. Hence, we had to keep our iterations low in our working results.

We also implemented fast search, however due to long resource allocation wait time, we weren't able to deduce results for the project in time.

7 ALTERNATE APPROACHES

Working with implicit feedback was definitely challenging. Initially, we experimented with various methodologies to develop an effective music recommendation system. Initially, we explored a non-normalized approach, which resulted in a relatively low root mean square error (RMSE) of approximately 5.58. However, this modeling approach exhibit inefficiencies. Subsequently, we implemented the implicit.AlternatingLeastSquares() method of implicit library, specifically designed for recommendation system based on implicit feedback. Additionally, we constructed a CSR matrix to facilitate the process. Unfortunately, our efforts were impeded by insufficient space errors, preventing the completion of this particular approach.

8 CONTRIBUTIONS

Vaishnavi and Drishti focused on preprocessing, baseline, ALS and report writing. Pranjal worked on both the extensions as well contributed to the report.

9 REFERENCES

[1] https://youtu.be/FgGjc5oabrA

[2] https:://towardsdatascience.com/building-a-collaborative-filtering-recommender-system-with-clickstream-data-dffc86c8c65

[3] https://www.researchgate.net/publication/220765111_Collaborative_Filtering_for_Implicit_Feedback_Datasets

[4] https://listenbrainz.org/about/ (Dataset)

[5] Baseline model: https://github.com/nyu-big-data/final-project-group-29/blob/main/baseline.py

[6] Data exploration and preprocessing: https://github.com/nyu-big-data/final-project-group-29/blob/main/final_project.ipynb

[7] ALS: https://github.com/nyu-big-data/final-project-group-29/blob/main/AlternatingLeastSquare.ipynb

[8] LightFM: https://github.com/nyu-big-data/final-project-group-29/blob/main/LightFm1.ipynb

[9] FastSearch: https://github.com/nyu-big-data/final-project-group-29/blob/main/fastsearch1.py