Implicit Feedback Modeling for Music Recommendation

DS-GA 1004 Final Project Report

Chenqin Yang cy1355@nyu.edu
Jiayao Liu jl9875@nyu.edu
Zhengyang Bian zb612@nyu.edu
Center for Data Science, New York University

Abstract

In this project, we utilized the Spark engine to build and evaluate a music recommender system. Relying on implicit feedback modeling, we conducted experiments on building the baseline model and did hyperparamter tuning on the rank of latent factors, regularization parameter and the scaling for handling implicit feedbacks. Further, we evaluated several modification strategies on implicit feedback data and the efficiency gain achieved on accelerated query search from utilizing spatial data structure by using the Annoy package.

1 Introduction

In this information-rich era, how to efficiently retrieve key information and discover the most relevant contents give the birth of recommender system. By combining the traditional search with informative histories of users, recommender system has now been developed to bring personalization into consideration. To ensure the quality and relevance of the provided recommendations, big data has been serving as the key driving-forces behind the recommender system.

In this project, we intend to build a latent factor model to decompose the user-item interaction matrix into a product of user factor matrix U and item factor matrix V that consist of embeddings for users and items in a common vector space.

In the provided dataset, count measures how many times a user listens to a certain track. Instead of explicit feedback such as rating, count is considered as implicit feedback associated with the confidence level of observed user preference. A binary variable p_{ij} is introduced to indicate the preference of user$_i$ to item$_j$. For example, if the count value between user$_i$ and track$_j$ is greater than 0, it indicates that user$_i$ favors track$_j$.

To quantify the confidence level in observing the preference, we use a variable c_{ij} which is linearly correlated with implicit feedback values by a constant alpha. Generally speaking, the larger the count (R_{ij}) is, the more evidence and stronger belief that user$_i$ likes the track$_j$. Regularization term is also included in the objective function to avoid overfitting, the power of which is controlled by λ that is determined by hyperparameter tuning.

$$\min_{U,V} \sum_{(i,j)\in\Omega} c_{ij}(p_{ij}-\langle U_i,V_j\rangle)^2+\lambda\left(\sum_i\|U_i\|^2+\sum_j\|V_j\|^2\right) \quad \text{where } p_{ij}=\begin{cases} 1 & R_{ij}>0 \\ 0 & R_{ij}=0 \end{cases}, c_{ij}=1+\alpha R_{ij}$$

Alternating Least Squares (ALS), an iterative algorithm, is applied to generate the user and item embedding matrices. After initializing U and V, at each iteration, we fix one matrix and solve the other matrix by least squares. Next, the newly-updated factor matrix is held constant to optimize the other matrix. The iterative steps will be repeated until convergence.

2 Implementation

Dataset Our training dataset contains full histories for 1 million users and partial histories for 11 thousand users. The validation and test set consist of the remaining partial histories for those 11 thousand users. Each row in the dataset represents one interaction between the user and item. Namely, the datasets consist of three columns — `user_id`, `count`, `track_id` where `count` is an integer always greater than 0.

Data Preprocessing For this project, we used PySpark to preprocess the data and built the model. Because of the limited computing resources, we chose to downsample the previous 1 million users histories in the training dataset to 20% and appended the partial histories for 11 thousand users together as our new training set. We also employed `StringIndexer` to transform the columns into indicies before feeding them to the ALS model.

ALS To estimate the user and item matrices, we used the implementation of ALS from Spark's machine learning library MLlib (pyspark.ml) that can run in parallel. After fitting the ALS model, we made top 500 item predictions for on the validation and test set and retrieved the latent factors for users and items.

Code Availability The codes are available in the repository with a usage guide.

3 Evaluation

Let's first define M as the number of users[1]: $U = \{u_0, u_1, \ldots, u_{M-1}\}$; Each user (u_i) has a set of N ground truth relevant documents: $D_i = \{d_0, d_1, \ldots, d_{N-1}\}$; And a list of Q ordered recommended documents based on their relevance: $R_i = [r_0, r_1, \ldots, r_{Q-1}]$ Two evaluation metrics were selected evaluate the model performance. The core idea of both metrics is to compare the true relevant items list with the model recommended items list so that we can quantify the effectiveness of the model.

Precision at *k* This is a measure of how many of the predicted items are truly relevant, i.e. appear in the true relevant items set, regardless of the ordering of the recommended items. The mathematical definition is the following:

$$p(k) = \frac{1}{M} \sum_{i=0}^{M-1} \frac{1}{k} \sum_{j=0}^{\min(|D|,k)-1} \text{rel}_{D_i}\left(R_i(j)\right) \quad \text{where } \text{rel}_D(r) = \begin{cases} 1 & \text{if } r \in D \\ 0 & \text{otherwise} \end{cases}$$

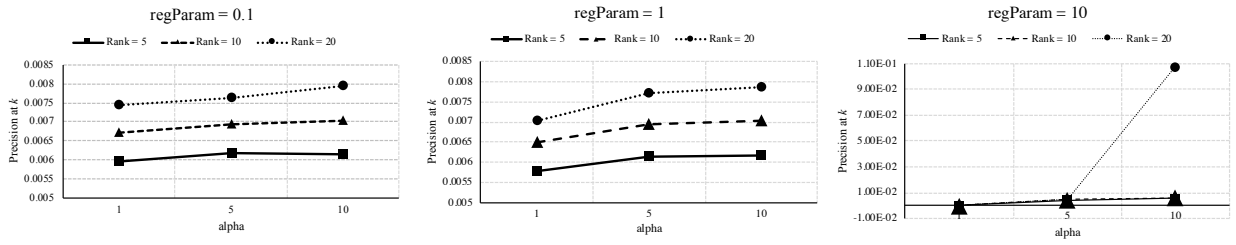
In this project, we will use k = 500, which means we will evaluate our model based on the top 500 recommended items.

Mean Average Precision (MAP) This is also a measure of how many of the predicted items also appear in the true relevant items set. However, the MAP score accounts for the order of the recommender. It will impose higher penalty for highly relevant items not being recommended with high relevance by the model, *i.e.* the item appears near the end of recommender list or even doesnt appear in the list. The mathematical definition is the following:

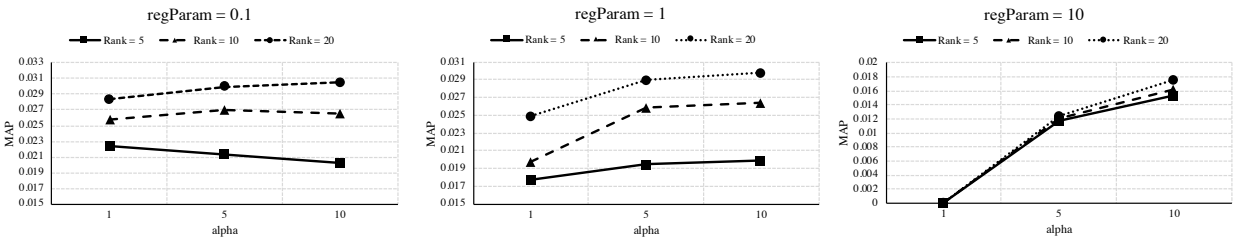
$$\text{MAP} = \frac{1}{M} \sum_{i=0}^{M-1} \frac{1}{|D_i|} \sum_{j=0}^{Q-1} \frac{\text{rel}_{D_i}\left(R_i(j)\right)}{j+1}$$

4 Results

Hyperparameter Search We performed a grid search over the following hyperparameters. The graphs of their Precision at k and MAP are shown in the figure below.



(a) Precision at k



(b) MAP

[1]The definition here follows the documentation of Apache Spark.
See https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html

- `rank`: [5, 10, 20] - the dimension of latent factors
- `regParam`: [0.1, 1, 10] - the regularization parameter to control overfitting
- `alpha`: [1, 5, 10] - the scaling parameter for handling the count data

The best model based on the Mean Average Precision score and Precision at k score we achieved happened to have the same hyperparameter settings. The **best setting** on the validation set is `rank` = 20, `regParam` = 0.1, `alpha` = 10. The corresponding MAP score is 0.0305 and Precision at 500 score is 0.0079.

Test set evaluation After hyperparameter tuning, we also evaluate our model on the **test set**. The results are listed below.

Model	MAP	Precision at k=500
`Baseline Model`	0.0305	0.0079

5 Extensions

5.1 Extension 1: Alternative model formualtions

In this section, we adopted two modification strategies, including log compression on count and dropping low count values to evaluated their impacts on the best baseline model.

Log Compression We replaced the count with the log of it (count $\rightarrow \log(1 + \text{count})$). From the statistics, it can be found that the distribution of count is heavily left skewed as more than half of the data take value 1, 75 percentile is 3 and the maximum count reaches 9667. Log transformation on count will reduce the skewness and variability of data.

Count	Mean	StdDev	Min	25%	50%	75%	Max
49824519	2.8762	6.4567	1	1	1	3	9667

Drop Low Counts In explicit feedback setting, the user will explicitly express their preference on items. However, in implicit feedback setting, the count will only serve as an indication of user preference.

For instance, if user_i listens to track_j for one time, user_i may accidentally click the play button or the user_i dislikes the track_j and never comes back to it while in the model, p_{ij} will be assigned 1 that indicates user_i favors track_j. Therefore, we dropped low count values for they may mislead the model by mistaking no or low preference for high preference.

Impact on Overall Accuracy The Precision at k and MAP of validation set are displayed below. It is found that the sole log compression modification produces higher Precision at k and MAP score compared to the baseline model while dropping low count values degrades the model performance to some extent. One explanation can be that log transformation alleviates the data skewness so that the latent factors calculated become more representative. On the other hand, dropping low counts leads to information loss, which may worsen the representativeness of latent factors.

Methods	MAP	Precision at k
`Log Compression`	0.0325	0.0082
`Drop Count = 1`	0.0233	0.0068
`Drop Count = 2`	0.0213	0.0065
`Log Compression + Drop Count = 1`	0.0219	0.0071
`Log Compression + Drop Count = 2`	0.0204	0.0066

Generalization To evaluate our model generalization performance, we apply the best baseline model and the best model from extension 1 on the test set. The performance is shown below:

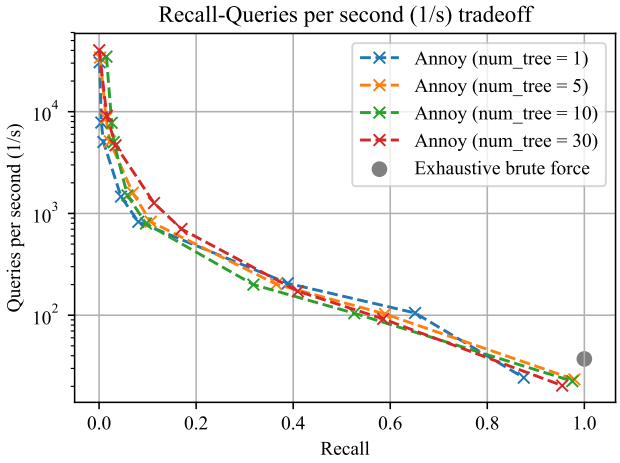
Model	MAP	Precision at k
`Baseline Model`	0.0305	0.0079
`Baseline Model + Log1p`	0.0313	0.0081

5.2 Extension 2: Fast search

Spatial trees utilize recursive partitioning on dataset to accelerate the searching speed for each query by reducing the time complexity from $O(n)$ to $O(\log n)$. In this project, we used Annoy package to achieve the query acceleration.

To begin with, the latent factors of users and items generated by the best model are exported to the local machine from Dumbo. Then, we searched the nearest item neighbours for a user query based on these factors using brute-force method and the Annoy package. Annoy recursively partitions all the points by randomly selecting splitting hyperplanes, formulating binary trees whose internal nodes represent a splitting hyperplane to split the points into two subspaces.

In terms of the brute-force method, we generated the list of top 500 recommended items by computing the dot products between one user and each item, which would be ranked by values to determine item relevance. The recommendation results are regarded as ground-truth, which will serve as the benchmark to calculate recall scores when we use the Annoy package.



When utilizing the Annoy package, we also did a comparison of the number of trees built in the `AnnoyIndex`. Each point on the graph is taken at different `search_k` values which represent the number of total nodes searched.

From the plot, there are several things worth analyzing:

- When the number of trees is fixed, we can observe the tradeoff between time and accuracy. In order to achieve higher accuracy, we have to sacrifice the query time.
- We can see that with the increase in number of trees built, the recall score achieved at the same `search_k` value generally increases. It is also noted that the query search time almost remains the same for different number of trees. Therefore, under similar time constraints, it is suggested to build more trees though it requires more memory.

Efficiency Gain The point on the lower right part of the plot from Annoy takes longer time than the brute-force method. One possible explanation is that when the `search_k` value is large, Annoy is close to exhaustive search but introduces additional overhead caused by searching through the nodes to form the nearest neighbour candidate set. Though the `search_k` value is greater than the total number of items (around 400,000), Annoy is still not equivalent to exhaustive search since it will loop through not only the leaf nodes but also the intermediate nodes. Therefore, the recall score is not exactly equal to 1.

In general, Annoy accelerates query search by restricting to a candidate set with a limited number of nearest neighbours so that we don't need to compute the dot products for all user-item pairs as in the brute-force method. For instance, we can achieve a three times faster query speed than brute-force at recall of 0.6. Therefore, if a compromise in recall is bearable, Annoy will bring time efficiency in nearest neighbour search. Overall speaking, the tradeoff between accuracy and time should be properly evaluated when we apply Annoy.

Contributions and Collaborations

Chenqin Yang: Baseline model, Extension 2, Report
Jiayao Liu: Baseline model, Extension 1, Report
Zhengyang Bian: Baseline model, Extension 2, Report