Recommender System for Music

Zihao Guo
First Year Master
NYU Center For Data Science
`zg866@nyu.edu`

Abstract

Recommender system has been widely used in generation of music, video and other commodity recommendation. We deploy the Alternating least squares (ALS) method in Spark to learn latent factor representations for users and items. For the baseline model, the best hyperparameter setting is “regParam” = 0.01, “rank” = 5, “alpha” = 0.01, which yields MAP (Mean Absolute Precision) 0.0293 and Precision at 500 0.01. For the model with count transformed, there is no obvious improvement from the baseline.

1 Introduction

Recommender system has 2 common types: collaborative filtering or content-based filtering. Collaborative filtering involves building a model from a user’s past behaviour, such as items previously selected and/or numerical ratings or count (implicit) associated with those items. Also, it takes similar users’ history into that user’s recommendation. Content-based filtering deploys a group of unique, predefined characteristics of an item so that items with similar properties will be recommended. Each system has its advantage and disadvantage. Collaborative filtering encounters cold start problem and content-based filtering encounters limited scope of properties, for example. In this project, collaborative filtering will be implemented because we have a decent amount of information about items for each user, so cold-start problem could be alleviated.

Whenever people listen to music on YouTube, Spotify or any other online platform, a list of recommended songs will be provided. The recommendation is usually determined by similar-taste users’ ratings of songs. However, most of the time, people don’t rate songs, so there is no explicit feedback of songs from those users. However, one can “estimate” the ratings based on the number of times that songs are listened to. The count is referred to the latent factor and the model would be a latent factor model, which is the focus of this project.

2 Data

2.1 Raw Data

The data are stored as parquet files. There are 4 columns in data: user_id, count, track_id, __index_level_0__. There are 3 data, namely train, validation and test set. Train contains full histories for approximately 1 million users, and partial histories for 110,000 users, located at the end of the train file. Validation contains the remainder of histories for 10 thousand users, which is used to tune the model. Test contains the remaining history for 100K users, which is used for final evaluation. A sample of raw data:

user_id	count	track_id	__index...
2c42e6551311710ca...	1	TRTCWHU12903CCE492	1722
77cac5c3389010b64...	3	TRZDLXJ128F9335C76	2996
3f9ed694a79835c92...	1	TRSPEFK128E07811EF	3582

2.2 Data Transformation

Since the user_id and track_id are both strings that are not accepted by the model (discussed in the next section), we transform them into numerical index representations. Also, in order to make sure the model works, downsampling at the first place is required. Parameter “handleInvalid” was set to “skip”, which automatically extract users from validation set that were already in the downsampled data. A sample of transformed data:

user_index	track_index	count
17562	7894	1
41949	1208	3
2595	15265	1

3 Model

As discussed in the Introduction session, collaborative filtering will be the choice of model. The method for collaborative filtering in this project is alternating least square. The alternating least squares (ALS) algorithm factorizes a given matrix R into two factors U and V such that $R \approx U^T V$. The unknown row dimension is given as a parameter to the algorithm and is called latent factors. Since matrix factorization can be used in the context of recommendation, the matrices U and V can be called user and item matrix, respectively. The i-th column of the user matrix is denoted by u_i and the ith column of the item matrix is v_i. The matrix R can be called the ratings matrix with $(R)_{i,j} = r_{i,j}$.

We are minimizing the following:

$$\begin{aligned} \arg\min_{U,V} \sum_{\{i,j|r_{i,j}\neq 0\}} \left(r_{i,j} - u_i^T v_j\right)^2 \\ + \lambda \left(\sum_i n_{u_i} \|u_i\|^2 + \sum_j n_{v_j} \|v_j\|^2 \right) \end{aligned} \quad (1)$$

with λ being the regularization factor, n_{ui} being the number of items the user i has rated and n_{vj} being the number of times the item j has been rated. This regularization scheme to avoid overfitting is called weighted-λ-regularization.

By fixing one of the matrices U or V, we obtain a quadratic form which can be solved directly. The solution of the modified problem is guaranteed to monotonically decrease the overall cost function. By applying this step alternately to the matrices U and V, we can iteratively improve the matrix factorization.

The matrix R is given in its sparse representation as a tuple of (i,j,r) where i denotes the row index, j the column index and r is the matrix value at position (i,j).

We make recommendation for a user based on "similar" items he/she would listen to. The score for each item to a user is $score = U_i \dot{V}^T$.

The recommendation model uses Spark's alternating least squares (ALS) to learn latent factor (count) representations for user and items. The models has some hyper-parameters, three of which will be optimized on, namely rank, regularization parameter (regParam) and alpha. The rank is the number of features to use (also referred to as the number of latent factors). The regParam is the regularization parameter to prevent overfitting. The alpha is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations (defaults to 1.0).

4 Evaluation

4.1 Model Implementation

First, I fit ALS model on the transformed train data. Since we don't have explicit rating, I set the parameter "implicitPrefs" to True. Then, I get the top 500 recommended items for validation users, which would be our prediction list. Then, we group validation data by user index and then aggregate the tracks for each user to get our true list. Finally, I join those 2 lists on the same user_index to generate a PredictionAndLabels RDD for the evaluation later.

4.2 Evaluation Metric

I deploy two evaluation metrics to evaluate performance of model. The first one is precision. Precision at k is a measure of how many of the first k recommended documents are in the set of true relevant documents averaged across all users. In this metric, the order of the recommendations is not taken into account. The formula is defined as:

$$p(k) = \frac{1}{M} \sum_{i=0}^{M-1} \frac{1}{k} \sum_{j=0}^{min(Q_i,k)-1} rel_{D_i}(R_i(j)) \quad (2)$$

The second one is Mean Average Precision (MAP). MAP is a measure of how many of the recommended documents are in the set of true relevant documents, where the order of the recommendations is taken into account (i.e. penalty for highly relevant documents is higher). The formula is defined as:

$$MAP = \frac{1}{M} \sum_{i=0}^{M-1} \frac{1}{N_i} \sum_{j=0}^{Q_i-1} \frac{rel_{D_i}(R_i(j))}{j+1} \quad (3)$$

4.3 Hyper-Parameter Tuning

I use both evaluation measures to assess model performance. I tune 3 hyper-parameters discussed in Model section on validation set to pick the best model configuration. Due to the limited resources for training model and large train data size, I have to tune the model on a much smaller dataset. Here we train on 10% train set. I initially set "regParam" = [0.001, 0.01, 0.1], "rank" = [2, 5, 10, 30, 50], “alpha” = [0.01, 0.1, 0.5, 0.8]. I grid search over the results and zoom in parameters when possible.

4.4 Result

The partial results after parameters zooming in and out are as follows:

RegParam	0.001	0.01	0.1
Rank	2	2	2
Alpha	0.01	0.01	0.01
MAP	0.009	0.012	0.011
Precision at 500	0.0021	0.0027	0.0029

RegParam	0.01	0.01	0.01
Rank	5	10	20
Alpha	0.01	0.01	0.01
MAP	0.013	0.012	0.012
Precision at 500	0.0034	0.0031	0.0030

RegParam	0.01	0.01	0.01
Rank	5	5	5
Alpha	0.001	0.05	0.1
MAP	0.011	0.012	0.012
Precision at 500	0.0035	0.0037	0.0034

I pick configuration “regParam” = 0.01, “rank” = 5, “alpha” = 0.01 as the optimal configuration after tuning. Then, I fit on this model setting on full train dataset, which yields “MAP” = 0.0281 and “precision at 500” = 0.01 on test set.

5 Extension

I discover there are 60% data in train having count = 1, which means those songs are only listened once by that user. However, common sense tells us count = 1 doesn’t necessarily yield a like or dislike, because the user likes it but forgets its name and can’t find it, or the user hates it and just listened to it once. Also, some counts are really large, which can be misleading, because a count = 100 and a count = 30 probably all indicates really “like”, but the model would treat them differently.

5.1 Drop Low Count

I drop the count = 1 from train data. Then, I run tuning again on 10% data again. The partial result is as follows:

RegParam	0.01	0.01	0.01
Rank	2	5	10
Alpha	0.001	0.01	0.001
MAP	0.012	0.012	0.011
Precision at 500	0.0032	0.0032	0.0031

As we can see, there is no improvement from the original setting, so this dropping low count strategy is not optimal.

5.2 Log Compress Data

Then, I log transform count: $log(1 + count)$. Then similarly, I rerun the model with various parameter settings, which yield the following result:

RegParam	0.01	0.01	0.01
Rank	2	5	10
Alpha	0.001	0.01	0.001
MAP	0.011	0.009	0.010
Precision at 500	0.0027	0.0029	0.0027

Still, we don’t see obvious improvement, so we also abandon this approach.

5.3 Hybrid Approach

I combine two approaches above to see how it will work. Since training cluster has been experiencing the breakdown, I’m only able to train on 1% data, which yields MAP = 0.0084. This score could be improved if we train on a larger portion of data. Thus, even though I don’t get an improvement, the result is inconclusive.

6 Appendix

6.1 Github Link

All codes can be found on **Github Account**

6.2 Contribution

I do this project by myself.

References

[1] Evaluation Metrics - RDD-based API
https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html.

[2] Collaborative Filtering.
https://spark.apache.org/docs/latest/ml-collaborative-filtering.html.

[3] ALS
https://medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe

[4] Pyspark
https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.recommendation.ALS

[5] Alternating least squares
https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/als.html