ALS Implicit Collaborative Filtering on Song Recommendation
DSGA1004 Final Project

Bin Zou, Weiyang Wen

Center for Data Science, New York University
May 2019

Abstract

We present a way to apply ALS implicit collaborative filtering in the domain of song recommendation under `PySpark`. We show the evaluation results in terms of ranking metrics with respect to different parameter settings. As an extension, we also discuss alternative model formulations to improve the ranking results.

1 Introduction and Motivation

Recommender system is an extensive class of web applications that involve predicting user responses to options. Recommender systems are widely utilized in a variety of areas. We can classify these systems into two broad groups: Content-based systems, such as recommendations for social media platforms Facebook and Twitter; Collaborative filtering systems recommend items based on similarity measures between users and/or items, such as playlist producers for music and video services YouTube and Pandora.

Recent research has shown that combining collaborative and content-based recommendation can be more effective. This hybrid method can be used to overcome the common problems in recommendation systems such as cold start and the data paucity problem.

Music recommender system has experienced a boom in recent years. Thanks to the online streaming services, users are granted access to almost all music around the world nowdays. Music recommender systems have achieved great success in finding interesing music to fit the users' preferences by filtering this abundance of music in advance.

Given the importance of the recommender system and its wide applications in the music industry, we're motivated to implement and evaluate a song recommender system using ALS implicit collaborative filtering by integrating various techniques we've learned in this class, including Hadoop, Spark, and storage optimization etc.

2 Implementation

Our main program consists of 3 components. Firstly, an `Indexer` function builds a pipeline to convert `user_id` and `track_id` into numerical values. Secondly, a `Trainer` function contains the main function of ALS, and lastly, a `Tester` applies the trained model to a test or validation set in order to obtain the ranking results. The details of each component are illustrated in the following sub-sections.

2.0 Preparatory Step

Since our dataset is large, we explicitly request more memory resources for the executor and the driver in order to prevent out-of-memory error. This is done by setting the `SparkConf` object, which stands for Configuration for a Spark application, before creating the spark session. Specifically, we set `executor.memory = 8G` and `driver.memory = 8G`.

2.1 Indexer

We firstly sample 20% of the training data to improve training speed while keeping the last 110K users for validation and testing. Sampling is done based on users instead of records.

Then a pipeline of string indexers to transform users and tracks are applied to transform the 3 sets of data.

```
Pipeline(stages=[user_indexer,
    track_indexer])
```

The formatted parquet are saved for succeeding testing.

2.2 ALS and the Trainer Function

In the train function, we utilize the alternating least squares (ALS) algorithm to learn these latent factors. There are a couple of parameters we would like to discuss here:

- `Rank`
- `regParam`
- `alpha`
- `implicitPrefs`

`Rank` refers to the number of features to use (the number of latent factors). In general, the higher the rank, the better the model's ability to describe the data(up to some point). However, higher rank model requires more memory and computation time to train. It is impossible to know how many underlying factors in advance but we can determine the optimal rank by experimentation. We start with a rank of 20, then increase by 20 at a time until the results stop improving.

`regParam` specifies the regularization parameter in ALS. It controls the L2 regularization scheme to help prevent over-fitting. We test the values from 0 to 10.

`alpha` is a constant controlling the rate of increase of the confidence matrix. Put it simply, we have different confidence levels with regard to play counts. As count increases, we have a stronger indication that the user indeed likes the song. Let r_{ui} be the play counts for user u and song i, then:

$$p_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases} \quad (1)$$

$$c_{ui} = 1 + \alpha r_{ui} \quad (2)$$

where c stands for confidence.

`implicitPrefs` Setting this to `True` tells spark.ml to deal with implicit feedback Essentially, the numbers are treated as user preferences rather than explicit ratings.

In summary, the ALS setting is:

```
als = ALS(rank=rank, maxIter=10,
    regParam=regParam, alpha=alpha,
    implicitPrefs = True,userCol="
    user_id_numeric", itemCol="
    track_id_numeric",ratingCol="count",
    coldStartStrategy="drop")
```

2.3 Tester

`Tester` is a module to do validation or testing. Firstly, an ALS model is loaded with `ALSModel.load` function. Then recommendation (predicted ranking) is generated using the following commend:

```
targetUsers = df_test.select("
    user_id_numeric").distinct()
userRecs = model.recommendForUserSubset(
    targetUsers,K)
```

where $k = 500$ is the number of items to recommend.

Before we can calculate ranking metrics, the next step is to generate $predictionAndLabels$, which is an RDD of (predicted ranking,ground truth set) pairs. Here we need to generate ground truth set from validation/test set. The ground truth set is an ordered list of track_id based on counts, group-by individual users. To achieve this, we use the $Window.partitionBy$ function to group tracks by users, and use the $orderBy$ function to order them by descending order of count. Then we convert it into a list for each user by using $F.collect\_list$. The functions are imported from *pyspark.sql* module. Putting it together:

```
w = Window.partitionBy("user_id_numeric"
    ).orderBy(df_test['count'].desc())

labels = df_test.withColumn('
    ActualRanking',F.collect_list("
    track_id_numeric").over(w))

labels = labels.select(['user_id_numeric
    ','ActualRanking']).dropDuplicates([
    'user_id_numeric'])

predictionsAndlabels = userRecs.join(
    labels, [labels.user_id_numeric==
    userRecs.user_id_numeric],'left').
    select('track_id_numeric', '
    ActualRanking')
```

Ranking metrics is used to quantify the effectiveness of these rankings or recommendations. Denote D the ground true set and $rel_D(r) = \begin{cases} 1 & \text{if } r \in D \\ 0 & \text{otherwise} \end{cases}$ where r is the recommended document. Here we report 3 kinds of metrics:

precision-At K counts how many of the first K recommended songs are in the true set. The order of the recommendations is not taken into account. We report the precision at 10 and at 500 respectively. Mathematically:

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

Mean Average Precision measures how many of the recommended songs are in the true set with the order of the recommendations taken into account:

$$MAP = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{|D_t|}\sum_{j=0}^{Q-1}\frac{rel_{D_t}(R_t(j))}{j+1}$$

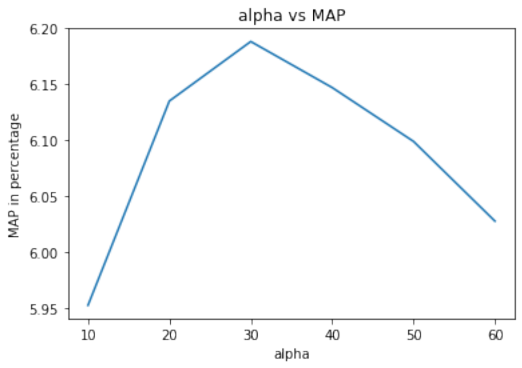
Normalized Discounted Cumulative Gain (nDCG) measures how many of the recommended songs are in the true set averaged across all users. It also takes into account the order of the recommendations:

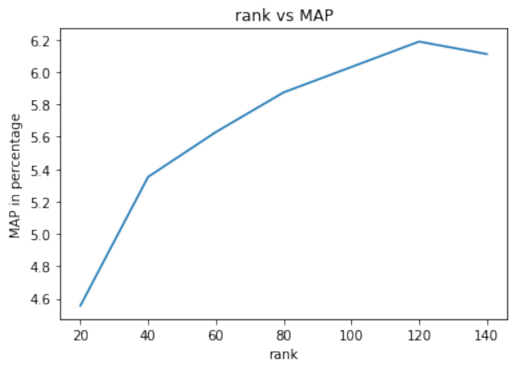
$$N(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{\text{IDCG}(D_1,k)}\sum_{j=0}^{n-1}\frac{rel_{D_i}(R_i(j))}{\ln(j+1)}$$

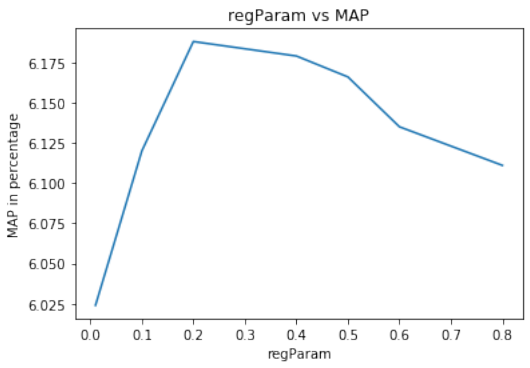
where $n = \min\left(\max\left(|R_i|, |D_i|\right), k\right)$ and $\text{IDCG}(D,k) = \sum_{j=0}^{\min(|D|,k)-1}\frac{1}{\ln(j+1)}$.

3 Evaluation Results

We do grid search on `rank`,`regParam` and `alpha`. The `ranks` we test ranging from 1 to 200, `regParam` ranging from 0 to 10, and `alpha` ranging from 0.01 to 60. Below we show the accuracy vs different values of parameters:







From these figures, we find that the optimal parameters are $(120, 0.2, 30)$. This number is consistent with the paper by Hu et al.(2008) where they report that optimal alpha is 40 and rank range between 100 and 200. The ranking metrics of validation set under optimal parameters are listed as following:

```
----R:120-reg:0.20-alpha:30.0-----------
        p(10)        2.39%
        p(500)       0.21%
        MAP          6.19%
        nDCG(500)    15.25%
----VALIDATION-SET---------------------
```

And the ranking metrics of test set under optimal parameters are as follows:

```
----R:120-reg:0.20-alpha:30.0-----------
        p(10)        2.32%
        p(500)       0.21%
        MAP          6.01%
        nDCG(500)    14.91%
----TEST-SET--------------------------
```

4 Extension

As an extension, we try two alternative model formulations. Firstly, we drop the low count values (counts = 1) in our data-set as we believe that listening to a song only once does not reflect a person's preference on that song. It is possible that the song is just randomly played or user clicks the song by error.

On the other hand, we also believe that high counts does not contain too much information either. A song played 50 times is not necessarily preferred over a song played 40 times. Thus we apply log transformations to smooth the behavior of the extreme values.

Combining the above two extenstions together, we apply both drop-low-count and log-transformation to our data-set. Again we tune the three parameters (Rank, regParam and alpha) by grid search. The optimal parameters are $(200, 8, 40)$ in this case and below are validation and test performance:

```
----R:200-reg:8.00-alpha:40.0-----------
        p(10)        1.95%
        p(500)       0.13%
        MAP          6.20%
        nDCG(500)    14.05%
----VALIDATION-SET---------------------

----R:200-reg:8.00-alpha:40.0-----------
        p(10)        1.95%
        p(500)       0.13%
        MAP          6.15%
        nDCG(500)    13.86%
----TEST-SET--------------------------
```

The result from the model with extensions is on a par with the performance of the original model without extensions on the validation set. There are some improvements of MAP on testing set, which implies that the new model has better generalizability. One possible reason of this improvement for the alternative formulation is the faster training speed. 60% of the observations have a count value equal to 1. Dropping those values will significant increase training and testing speed and save disk space, without impacting the performance of the model.

5 Conclusion

We use Spark's alternating least squares (ALS) method to learn latent factor representations for users and items in the setting of song recommendation. Our result achieves a MAP of 6% with a small fraction of data. Our findings of optimal parameters confirm previous finding in publications (Zhou et al., 2008). For extension, drop 1 count and log transformation significantly decrease training time and achieve better generalizability.

Contributions: Weiyang is responsible for building Basic recommender system and Bin is responsible for building alternative model (extension). Equal contribution for both in parameter tuning and report write-up.

References

[1] Hu, Yifan, Yehuda Koren, and Chris Volinsky. *Collaborative Filtering for Implicit Feedback Datasets*, ICDM. Vol. 8. 2008.

[2] Zhou, Yunhong, et al. *Large-scale parallel collaborative filtering for the netflix prize." International conference on algorithmic applications in management*, Springer, Berlin, Heidelberg, 2008.