Recommendation System with Spark

Ravi Choudhary
rc3620@nyu.edu

Jatin Khilnani
jk6373@nyu.edu

1 Introduction

This project deals with building and evaluating a recommendation system using Alternating Least Squares (ALS) method (Koren et al., 2009). The data given to us is the interaction of more than a million users for certain music tracks. We learn latent factor representations for users & tracks through `spark.ml.ALS` (Apache) to recommend music tracks for 10% of the users in the test set. We then explore and evaluate different modification strategies to change the default behavior of the model and improve the recommendation system.

2 Data

2.1 Data Understanding

Base data (to build the utility matrix) for the recommendation system is provided in three sets, namely train, validation and test. Training data has complete interaction history for 1M users, and partial history of 110k users. Remaining partial interactions of these users are available in sets of 10k & 100k as validation and test sets respectively. Each interaction record in the data set has a User ID, Count of Interactions and Track ID.

User ID & Track ID are hexadecimal strings representing unique users and tracks respectively. The counts are all non-negative integers, indicating that the data has only complete interactions or partial interactions have either been removed or also been assigned integers.

Supplementary data sets containing track metadata, features, tags and lyrics are available in a separate file each. There are a total of 1M unique tracks for which the metadata is available.

2.2 Data Preparation

Since ALS in Spark ML requires the User ID and Track ID to be numeric, our first step is to transform the data sets for this purpose. We employ `StringIndexer` trained on base training data for users and track metadata for trackid. We use track metadata since it contains full list of tracks for processing, while there are only 385,371 tracks in training interactions. Hence, using metadata would cover tracks present in validation and test set that might be missing in the training data, but would be required for ranking metric evaluation. We save all the three modified base data sets on HDFS for further work.

3 Methodology

This recommendation system is based on 'Latent Factor' model following *Collaborative Filtering* approach, which captures the users' past behaviour through the count of interactions available in the base data sets. Since these counts provide an implicit feedback of user preference, we set the `implicitPrefs=True` option present in the Spark ML ALS model. The implicit preference option adapts the methodology suggested by Hu et al. (2008).

3.1 Basic Recommendation System

We assume that as the number of users interacting with the tracks increases, we will get a better latent representation. Therefore, we train the base model on the entire training data set instead of sampling from the same.

We then tune this model on the validation set using `RegressionEvaluator`. As mentioned earlier, there might be a mismatch between tracks present in the train & validation set, because of which model would return `NaN` predictions for tracks which were not present/fit using the train set. This would lead to invalid results for evaluation metric as well, and hyper-parameter tuning would be difficult. In order to workaround this cold-start problem, we change the default behaviour of the model by setting the parameter `coldStartStrategy='drop'` for dropping `NaN` predictions.

After choosing the best configuration, performance of the model(s) was evaluated on the validation and test data using ranking metrics.

3.2 Hyper-parameter Tuning

While ranking metrics would have been a better option for selection of hyper-parameters as well, we decide to use Root Mean Square Error (RMSE) for tuning. The decision in favor of latter is made by taking into consideration that `RankingMetrics` function is only created for RDD and not for the Dataframe API which would have meant a lot of time spent on Dataframe to RDD conversion and ranking of all items.

We perform grid search on the hyper-parameters `rank` (dimension of the latent factors), `regParam` (regularization parameter) & `alpha` (scaling parameter for handling implicit feedback), and choose the model with the best (least) RMSE on validation data. Spark model internally scales the regularization parameter at each step based on number of interactions required for updating user/item factors using ALS-WR approach (Zhou et al., 2008).

3.3 Alternative Model Formulations (Extension 1)

As part of this extension, we experiment with both the modifications suggested that is, log compression and dropping low count interactions, to change the model behaviour and review the same.

For log compression, we use the formula $r_{new} = log(1 + r_i/\epsilon)$, where $\epsilon = 1$. We set ϵ to 1 because the interaction count is integer type and logarithm of count would reflect its scale.

Table 1: Number of Interactions by Count

Count	Number of Interactions	% Total
1	29,595,102	59.4
2	7,550,537	15.2
3	3,309,997	6.6
4	1,860,239	3.7
5	2,324,541	4.7
> 5	5,184,103	10.4

For dropping low counts, as shown in the table above, we observe that almost 60% of the interactions have count $= 1$. We drop these interactions to ascertain its impact on the model. We keep the threshold at one, assuming that repetition of an interaction means affinity towards the track, and also since that removes majority of the dataset.

4 Results and Analysis

4.1 Hyper-parameter Tuning

We restrict the range of hyper-parameters due to limitations on time and computations possible on the Hadoop cluster (Dumbo).

- Rank: We choose to use the rank $\in \{6, 9, 12\}$ for hyper-parameter tuning. Keeping all the other hyper-parameters same, increasing the rank resulted in best RMSE.
- Regularization: We experiment with values of $(0.001, 0.01, 0.1, 1)$. With other parameters constant, decreasing regularization gives best RMSE. Hence, we use values $10^{-5}, 10^{-4}$ for final experiments.
- Alpha: Keeping all the other hyper-parameters same, increasing the alpha resulted in best RMSE. For final experiments we keep alpha values in $\{10, 20, 30, 40\}$.

In next sections, we report the exhaustive tuning results with best result in each set **highlighted**.

4.1.1 Basic Recommendation System

Table 2: Validation RMSE

Rank	Regularization	Alpha	RMSE
6	1e-03	0.01	7.711494119688186
		0.1	7.709121570307001
		1	7.694977303170774
		10	7.651545231817701
6	1e-02	0.01	7.711637198703407
		0.1	7.7092008316732175
		1	7.694934098399986
		10	7.6521097905176925
6	1e-01	0.01	7.712260424478288
		0.1	7.709900638170995
		1	7.694879479818225
		10	7.652299992855418
6	1	0.01	7.7166808640161655
		0.1	7.716675405683557
		1	7.6997818443676636
9	1e-05	1	7.692539169104772
		10	7.645959044837932
		20	7.623781395860365
		30	**7.60894979728134**
		40	7.59771159461483
9	1e-04	1	7.692791168744366
		10	7.645989296904664
		20	7.623793782672034
		30	7.608957016156779
12	1e-05	1	7.690802187092469
		10	7.642376455470178
12	1e-04	1	7.690990092776905
		10	7.6424026875088495

4.1.2 Alternative Model Formulations

Table 3: Validation RMSE of log compression

Rank	Regularization	Alpha	RMSE
9	1e-05	10	1.21566343218087
		20	1.19009341676278
		30	1.17047914413803
		40	1.15462973122186
9	1e-04	10	1.21576426816943
		20	1.19020568532029
		30	1.17057055435472
		40	1.15469630328287
12	1e-05	10	1.21042588052752
		20	1.18220778758446
		30	1.16157431157943
		40	**1.14500121681744**
12	1e-04	10	1.21059386535144
		20	1.18231795061875
		30	1.16164764807776
		40	1.14505490364818

Table 4: Validation RMSE of dropping low count

Rank	Regularization	Alpha	RMSE
9	1e-05	10	7.892482785628
		20	7.87081889195913
		30	7.85660749428085
		40	7.84592794521806
9	1e-04	10	7.89250628819681
		20	7.87082739854682
		30	7.8566122409535
		40	7.84593109893315
12	1e-05	10	7.8888235189806
		20	7.86679678476004
		30	7.85242338739148
		40	**7.84164320800601**
12	1e-04	10	7.88884064987529
		20	7.86680328606551
		30	7.85242702999042
		40	7.84164562451125

4.2 Ranking Metrics

Based on the hyper-parameter tuning, we choose the best model in each segment and derive the Mean Average Precision, Precision and NDCG for the top 500 predictions of the model on validation and test data. `rank()` function from PySpark SQL is used to rank and obtain the ground truth of top 500 tracks for each user, which is peculiar in the sense that it gives same rank to the tracks with same count. This might result in some inaccuracies in cases where the user has interacted with more than 500 tracks and most of tracks have equal number of counts, as it may result in the ground truth having more than 500 tracks.
There are no such users in the test data, so we expect no such inaccuracy in the current result set. Due to time and resource constraints, some of the hyper-parameter combinations are not tried in all the models.

Table 5: Ranking Metrics – Mean Average Precision

Model Type	Data	Mean Average Precision
Basic Model	Test	0.029913185823079627
Basic Model	Val	0.02918501878415256
Log Compression	Test	0.03961446833533805
Log Compression	Val	0.03927348451046232
Drop Low Count	Test	0.0310260941815157
Drop Low Count	Val	0.030376404176050097

Table 6: Ranking Metrics – Precision @500

Model Type	Data	Precision @500
Basic Model	Val	0.008319000000000005
Basic Model	Test	0.008342180000000001
Log Compression	Val	0.009152199999999994
Log Compression	Test	0.009154439999999995
Drop Low Count	Val	0.008244997236042011
Drop Low Count	Test	0.008281294448426048

Table 7: Ranking Metrics – NDCG @500

Model Type	Data	NDCG @500
Basic Model	Val	0.14538858981226205
Basic Model	Test	0.14615660405165298
Log Compression	Val	0.16858650220748564
Log Compression	Test	0.16869423398280864
Drop Low Count	Val	0.14411017803795528
Drop Low Count	Test	0.1451889260663911

5 Discussion and Conclusion

The results depict that Log Compression model out-performs the Basic and the Drop Low Count version on validation data with a reduction in RMSE by almost a factor of 7 over both.
Results from ranking metrics indicate that Log Compression gives the best results with almost 33% higher mean average precision compared to the other 2 models for similar hyper-parameters. Though the combination of hyper-parameters that gives the best log-compression model was never tried for the basic model, the increase in mean average precision of this scale seems unlikely for the basic model even with the same hyper-parameters.

Future Work As seen above, we get the best results at the least or highest hyper-parameter values, which indicates further scope of tuning through search in either direction, using `RankingMetrics`. We can also utilize tracks metadata to identify similarity, perform item clustering and improve initialization.

References

Apache. Collaborative Filtering – Spark 2.4.3 Documentation.

Y. Hu, Y. Koren, and C. Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *2008 Eighth IEEE International Conference on Data Mining*, pages 263–272.

Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8):30–37.

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic Aspects in Information and Management*, pages 337–348, Berlin, Heidelberg. Springer Berlin Heidelberg.

Collaboration

Following are the contributions of the group.

- Ravi Choudhary:
  - Code for Basic and Log Compression based recommendation system
  - Code for indexing User ID & Track ID
  - Project Report
- Jatin Khilnani:
  - Code for Recommendation System for dropping low count
  - Code for Ranking Metrics Evaluation
  - Project Report

Project artifacts (code, results & report) are available on our GitHub repository.