A Rcommender System Using Implicit Feedback

Nyutian Long (nl1668) , Linfeng Zhang (lz1883)
NYU Center for Data Science

1 Introduction

Collaborative Filtering is a common and powerful way for building feedback-based recommender systems. In the big data regime, we could model bi-linear interactions between users and items by filling out the utility matrix, which is usually sparse. Alternating least square is a brief and straightforward algorithm to complete the utility matrix, but we do need to think about possible modification strategies of the implicit feedback. Unlike explicit feedback, implicit feedback often demonstrates weaker signals of uers' true preference to items. How do we extract as much information as possible by transformation the feedback data? How to efficiently deal with a large dataset? In our project, we focused on the modification strategies of the count data, the reasons behind and their impact on model performance. We also discussed technique difficulties we met dealing with large dataset, compromises we made and possible improvements in the future.

2 Previous Work

Extensive works on collaborative filtering have been done by researchers. Hu, Koren and Volinsky discussed how to deal with implicit feedback when building and evaluating a collaborative filtering recommender system.[1] The approach to deal with implicit feedback used in spark.ml.als was taken from this paper.

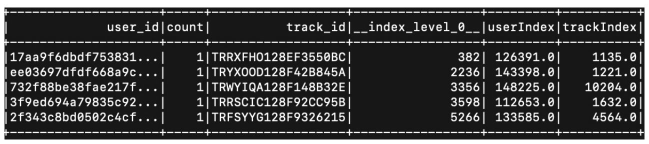
3 Methodology

In our project, we were provided with 7 datasets, cf_train, cf_validation, cf_test, metadate, features, tags and lyrics, from which we could build and evaluate a recommender system. We only used cf_train, cf_validation, cf_test. The training set contained complete history of 1 million users and partial history for $110,000$ users while the validation set and test contained remaining history for the $10,000$ and $100,000$ users. We used Apache Spark to preprocess data and train our models using distributed machine learning algorithms on NYU dumbo cluster, a 48-node Hadoop cluster. Our project implementation could be summarized as following: first we preprocessed the training, validation and test sets, after which we passed the training set to matrix factorization algorithms to learn embeddings of users and items into a common vector space. The validation set was used to tune hyperparameters using mean average precision of the top 500 items for each user. Same metrics were used to evaluate our model performance on the test set. Next, we explored variant ways of modifying the count data and analyzed their impact on data performance and possible reasons behind.

[1] *Collaborative Filtering for Implicit Feedback Datasets* by Hu, Koren, Volinsky

3.1 Data Preparation

As mentioned before, we used cf_train, cf_validation, cf_test to buid and evaluate a song recommender system. The schema was (user_id StringType, count LongType, track_id StringType, __index_level_0__ LongType). Since the alternating least square method provided by spark.ml only accepts numerical inputs for the user column and item column, we decided to use same StringIndex objects to transform the training date, validation data and test data, and then saved them as new parquet files on hdfs. The first challenge we met was to save the indexed training data where we kept getting a java heap space error. To solve this, our first approach was to change the data storage by repartitioning the indexed training data, which, unfortunately, didn't work. We tried to repartition by userIndex, trackIndex or count to 2000 or 5000 partitions. Neither worked. Our second approach was to downsample the training set. To do this, we first sampled every user that was in the validation set or test test. Without them, we would not be able to evaluate the model performance. At the same time, to avoid totally ignoring interactions of users not in the validation or test sets, our second sample consisted of 1 % user-song interactions. We then make a union of the two samples and ended up with 1500701 interactions, which was about only 3% of the original training data. We then saved our indexed training, validation and test date on hdfs. Our output data had the same schema as our input data. Figure 1 was some data instances of our final datasets. All codes associated with downsampling and indexing were in index.py. we used spark-submit hdfs:/user/bm106/pub/project/cf_train.parquet hdfs:/user/bm106/pub/project/cf_validation.parquet hdfs:/user/bm106/pub/project/cf_test.parquet ./train.parquet ./val.parquet ./ test.parquet to submit the spark application to save our output files.



```
+--------------------+-----+------------------+-----------------+---------+----------+
|             user_id|count|          track_id|__index_level_0__|userIndex|trackIndex|
+--------------------+-----+------------------+-----------------+---------+----------+
|17aa9f6dbdf753831...|    1|TRRXFHO128EF3550BC|              382| 126391.0|    1135.0|
|ee03697dfdf668a9c...|    1|TRYXOOD128F42B845A|             2236| 143398.0|    1221.0|
|732f88be38fae217f...|    1|TRWYIQA128F148B32E|             3356| 148225.0|   10204.0|
|3f9ed694a79835c92...|    1|TRRSCIC128F92CC95B|             3598| 112653.0|    1632.0|
|2f343c8bd0502c4cf...|    1|TRFSYYG128F9326215|             5266| 133585.0|    4564.0|
+--------------------+-----+------------------+-----------------+---------+----------+
```

Figure 1: Input Instances

3.2 Baseline

For our baseline model, we implemented Alternating Least Square algorithm using spark.ml.als without modifying count values. They way they dealt with implicit feedback was taken from *collaborative filtering have been done by researchers*[2]. Rather than directly model the utility matrix, this approach treats the data, r_{ui}, the observed value that user u gave to item i, as numbers representing the strength in observations of user actions, p_{ui}, which was calculated as follows.

$$p_{ui} \begin{cases} 1, & r_{ui} > 0 \\ 0, & r_{ui} = 0 \end{cases}$$

One thing to note here is that our dataset did not have count value that was 0. In the case, the p_{ui} was either 1 or an empty value to be predicted. Those numbers are then related to the level of confidence in observed user preferences, $c_{ui} = 1 + \alpha r_{ui}$, rather than explicit ratings given to items. The objective function to minimize was

$$\sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2).$$

We fitted 27 hyperparameter (rank, reg_parameter, alpha) combinations to the training set, and then evaluated the performance on validation set to find the best hyperparameter. Since the training set and validation set had been pre-defined, we could not use the built-in cross validation provided by Spark.ml. We decided to use itertools to optimize running time over nested for loops. Ideally, we wanted to search over a broad range of each hyperparameter and keep zooming in until we get the best hyperparameter combination. However, we have been encountering random failures on dumbo cluster, which took too much time. Considering that we still had to proceed to the extension, we had to make a compromise on the baseline model by choosing the best hyperparamter out of the only 27 combinations we tried. Codes for training the baseline model are in als.py. I used spark-submit als.py ./train.parquet ./val.parquet ./als to submit the application and saved the train model to hdfs.

3.3 Extension: Alternative Model Formulations

Aiming to improve the baseline model, we tried three modification strategies, log compression the count data, dropping values less than 2, and dropping values less than 3.

Log compression targeted to compress large values by mapping R_{ij} to $\log(1 + R_{ij})$. The underlying assumption is that numerical differences in larger values are not as informative as numerical differences in smaller values. The log compression also addresses the unique characteristics of implicit feedback:

- "Transferring the raw observations (r_{ui}) into two separate magnitudes with distinct interpretations: preferences (p_{ui}) and confidence levels (c_{ui}). This better reflect the nature of the data and is essential to improving prediction accuracy, as shown in the experimental study
- This algorithm addresses all possible (nm) user-item combinations in a linear running time, by exploiting the algebraic structure of the variables." (Hu, Koren and Volinsky)[3]

Dropping low count values changed the way r_{ui} was binarized. p_{ui} was now calculated as follows.

$$p_{ui} \begin{cases} 1, & r_{ui} > k \\ 0, & r_{ui} = 0 \end{cases}$$

The values below k would be dropped. However as mentioned before, our dataset actually did not have any user-item interaction of a 0 count. Now for count values greater than or equal to k, we would have $p_{ui} = 1$. Songs with count less than k became either an empty value to be predicted or removed from the system. Codes for training the log compressed count is in log_als.py. Codes for training the model with dropping count below 2 is in als_drop_1.py. Codes used for training the model with dropping count value below 3 is in als_drop_1.py. To train these models, we submitted spark applications as spark-submit xx.py ./train.parquet ./val.parquet ./xxmodel.

4 Evaluation

For the evaluation, we chose to use MAP@500 for both the training process and the evaluation on test data. Despite the fact that we were required to use ranking metrics for evaluating the test data, we personally believed that MAP was a better choice than regression metrics such RMSE and MSE in our case for two reasons. First, evaluation should model behavior. In a real-life setting, songs are typically recommend together as a list. And we would care more about whether users like the recommended songs much more than how many times users have listen to certain songs. Second, the numerical transformations done on implicit feedback made the final predicted numerical value hard to interpret. Considering that implicit feedback is informative, but hard to predict, spark.ml deals with implicit feedback as described above. Unlike a recommender system with explicit feedback, after all those transform, differences in numerical values shall not indicate users' true preference for songs as much as ranking metrcis. Since our test data contained 100,000 users, we encountered job failure twice after running the spark application to evaluate the test set for 5 hours. We thought that repartition the test set might help, so we changed the storage of test set, after which evaluating the test set only took 2 hours.

Table 1 (in the last page) contains all the MAP@500 on the validation set under different hyperparameters. As we can see, the MAP@500 reached maximum when rank = 40,

[2]3

[3]

regularization parameter = 0.1 and alpha = 40.

Table 2 (in the last page) contains the MAP@500 on the test set for the optimal hyperparameter combination in each model. As we can see, log compression improved the baseline MAP by 11.24%, but both dropping count value below 2 and dropping count value below 3 actually hurt the performance of our model.

The reason why log compression improved our model performance was easy to comprehend. For fairly large count values, the level of confidence in observed user preference may not differ very much, and log compression just corresponded with this. As for dropping low count values, we had expected our model to outperform the baseline, but it did the contrary. We thought there might be two reasons. First, a lot of the entries in our dataset had low count value; thus dropping low count value made our three dataset shrink. In that case, there might simply not be enough data left for us to build and evaluate our model. For example, after dropping count value below 2, our training set ended up with 631684 records, which was only 42% of the one used for baseline model. After dropping count value less than 3, we were left with 410585 entries. Second, it may simply be the the case that intrinsic properties of our dataset determined that binarizing p_{ui} at 1, rather than any other value k, would be the optimal choice. Codes associated with evaluating the test set is in test.py. To run, simply submit an spark application as spark-submit test.py ./als ./test.parquet arg[3] where arg[3] is an option ('log', 'drop1' or 'drop2') how you want to deal with count data in test set.

5 Conclusion

We built and evaluated a song recommender system using count feedback. To improve the baseline model, we tried log compression the count data, dropping count value below 2 and dropping count value below 3. Only log compression improved the baseline performance; dropping low count values actually jeopardized our model. Given the large dataset, a lot of time was taken to tune hyperparameters and evaluate test performances. Though we sub-sampled the training set to only 3 % of the original dataset, training the model with 27 hypterparameter combinations took around 8 hours and evaluating the test set performance took more than 2 hours on average if no random intermediate error occurred on dumbo cluster. For several times, our submitted spark application started running just fine and printed out results, but after a while, we kept getting error about not enough replicas or not enough workers. And the spark application stopped. With this technical constraint, we had to make some compromises not to dive into hyperparameter tuning too much. This compromise made us fail to zoom into smaller range to get the optimal hyperparameters.

If time and resources permit, our model could be improved in two ways. First of all, we would use the full sample of training set instead of a subsample. In our implementation, we were virtually ignoring 97% user-song interactions, which disabled us from capturing some common interaction patterns. Besides, when tuning the hyperparameter, we could try more combinations. We would first search over a broach range of hyperparameter. Once we get in a range where the derivative of the MAP curve changes direction, we keep zoom in until we find the optimal hyperparameter.

6 Contributions

Nyutian: subsample; baseline model; model with log cmopression on count; evaluated test performance
Linfeng: indexed the training, validation and test set and wrote parquet files on hdfs; wrote the script for training models after dropping the lowest count values; evaluated test performance

References

[1] George, T. and Merugu, S. (n.d.). *A Scalable Collaborative Filtering Framework based on Co-clustering.* [ebook] [Accessed 8 Dec. 2018].

[2] Tanna, K., Fevry, T. and Murat, V. (n.d.). *Datamufe - Project Report The Right Price.* [ebook] [Accessed 8 Dec. 2018].

[3] Selivanov, Dmitriy. "Matrix Factorization for Recommender Systems · Data Science Notes." *Data Science Notes*, 28 May 2017, dsnotes.com/post/2017-05-28-matrix-factorization-for-recommender-systems/.

[4] McNee, Sean M., et al. *Being Accurate Is Not Enough: How Accuracy Metrics Have Hurt Recommender Systems* . GroupLens Research, 2006, pp. 1097–1101, *Being Accurate Is Not Enough: How Accuracy Metrics Have Hurt Recommender Systems* .

[5] Hu, Yifan, et al. *Collaborative Filtering for Implicit Feedback Datasets. 2006, Collaborative Filtering for Implicit Feedback Datasets*, yifanhu.net/PUB/cf.pdf.

rank	**reg**	**alpha**	**baseline**	**log compression**	**drop below 2**	**drop below 3**
10	0.001	1	0.0265	0.0234	0.0184	0.0264
10	0.001	20	0.0284	0.0340	0.0155	0.0257
10	0.001	40	0.0279	0.0355	0.0136	0.0230
10	0.01	1	0.0265	0.0234	0.0183	0.0262
10	0.01	20	0.0284	0.0338	0.0156	0.0257
10	0.01	40	0.0280	0.0354	0.0136	0.0230
10	0.1	1	0.0273	0.0249	0.0183	0.0265
10	0.1	20	0.0288	0.0338	0.0163	0.0259
10	0.1	40	0.0282	0.0352	0.0132	0.0230
20	0.001	1	0.0279	0.0230	0.0273	0.0280
20	0.001	20	0.0343	0.0383	0.0316	0.0272
20	0.001	40	0.0345	0.0405	0.0301	0.0245
20	0.01	1	0.0278	0.0232	0.0272	0.0275
20	0.01	20	0.0339	0.0380	0.0312	0.0272
20	0.01	40	0.0338	0.0405	0.0301	0.0245
20	0.1	1	0.0284	0.025	0.0275	0.0277
20	0.1	20	0.0340	0.0378	0.0318	0.0272
20	0.1	40	0.0339	0.0401	0.0303	0.0246
40	0.001	1	0.0299	0.0229	0.0295	0.0293
40	0.001	20	0.0410	0.0434	0.0357	0.0332
40	0.001	40	0.0415	0.0461	0.0341	0.0309
40	0.01	1	0.0297	0.0229	0.0293	0.0298
40	0.01	20	0.0410	0.0431	0.0358	0.0331
40	0.01	40	0.0415	0.0461	0.0342	0.0308
40	0.1	1	0.0304	0.0257	0.0293	0.0297
40	0.1	20	0.0411	0.0425	0.0358	0.0331
40	0.1	40	0.0416	0.0456	0.0343	0.0308

Table 1: Evaluation on validation set

	rank	**alpha**	**alpha**	**MAP**
baseline	40	0.1	40	0.0418
log compression	40	0.001	40	0.0465
drop below 2	40	0.1	20	0.0373
drop below 3	40	0.001	20	0.0354

Table 2: comparison of different modification strategies on count