1004 RECOMMENDATION SYSTEM

NYU Center for Data Science

Chuan Chen
cc6580@nyu.edu

Yihang Zhang
yz2865@nyu.edu

May 12, 2020

1 Business Understanding

Recommendation has long been an essential feature of modern social life. Nowadays people are enjoying the convenience and satisfaction brought by those recommendation systems everywhere on their ends, no matter it is an mobile application or a streaming website. The recommendation system is making such an impact because it actively recommends with personalization rather than wait for users to search for items and information they are interested in. With taking advantage of processing user's past history through data science technology, recommendation system is able to understand user's preference by seizing the similarity between items consumed and between different users. Therefore, users are more prone to get the appropriate information related to current interests from recommenders, which indicates a more efficient distribution of community resources from providers to users. In one word, recommendation system is taking the place of user-searching as the main interactive method.

2 Data Understanding

The datasets are collected from goodreads.com in late 2017, acquired on UCSD Book Graph. Three datasets are used in this project, including interaction data, user id mapping data and book id mapping data. The interaction data, which is mainly used to construct the recommendation model, contains 229 million user-book interactions for each users. While the rest of two datasets are used to map book ids and user ids in interaction data file to their real ids. The interaction data includes 5 columns, recording each interaction's user_id, book_id, is_read(1:read, 0:unread), rating(0-5), is_reviewed(1:reviewed, 0:unreviewed). After having observed data distribution across different columns, we figured that there are noticeable amount of interactions giving 0 rating and some users are having few interactions. Therefore, we need to take them into consideration for model performance when training the model.

3 Data Preparation

To prepare the data, we retrieved 1% random interactions from the full data. We first read the whole data csv using pyspark on dumbo and used sql queries LIMIT to subset the 1% sample. However, we found that the sample we got in this way was not user-based since our sample should be interactions of 1% users. Hence, we used df.sample to randomly sample 1% user from user_id_map data file and then join with the interaction data file by matching user ids so as to construct our interactions sample. Then we tried to write this dataframe to parquet but it turned out that parquet file could not be successfully saved from hdfs to local directory. So we overwrote the data to csv file and saved it to local. Then on our local end, we randomly

subset 60% users' interactions as train set and two 20% samples as validation and test sets. After that, we moved half interactions of each user in validation and test sets back into the train set to assure that all we have information of all users in the train set.

To deal with the full data, we firstly uploaded our datasplit.py to dumbo and tried to run it by python but the task was constantly killed. Then we tried to run it through calling spark-submit alias but still failed due to the constant memory error. It seemed that such a huge data size outweighed Pandas capability. In this case, we decided to directly split data on dumbo using pyspark sql. Unfortunately, this method also failed because pyspark continuously stopped responding with error message 'ERROR BroadcastExchangeExec: Could not execute broadcast in 300 secs' once we were at the last step. To fix this problem, we also tried to overwrite the data file as parquet but the error constantly interrupted pyspark. It is extremely frustrating that we have not successfully splitted the full data till the end, which means that we only have the result trained from using the 1% data and tuned by partial data.

4 ALS model

After obtaining the train, validation, and test sets from the down-sampled data, we loaded them into spark Dataframes and kept only the 3 columns of interest, `user_id`, `book_id`, and `rating`. We tuned the two hyper-parameter values on the ALS model, the regularization coefficient and the number of latent factors, using values showed in table 1.

ranks	8	10	12	14	16	18	20
regs	0.001	0.01	0.05	0.1	0.2		

Table 1: parameter values used in grid search

The best model was selected based on predicting ratings that results in the lowest RMSE when compared to the true ratings in the validation set. Using the python shell on local machine, the entire parameter tuning process took 4046.15 seconds, and returned `reg=0.2` and `rank=20` as the best model parameters, giving a validation RMSE of 1.7961. The complete tuning outputs can be seen in figure 3. Then, the model learning curve is plotted testing number from 1 to 11 as the maximum number of iterations for the ALS model fitting. The total run time for learning is 1000.92 seconds, and results are shown in figure 1. As shown, the RMSE converges at about 4 iterations. A prediction was then made on the test dataset using the best obtained model, and the test RMSE is 1.7330. The first 5 rows of predictions are shown in figure 4. Then, the top-500 recommendations was made for all users. However, when we tried to visualize this dataframe using `.show()`, the run time took more than 2 hours, and ultimately prompted us to interrupt this execution.

Upon reviewing this table schema, shown in figure 5, we suspect that due to the complicated nested structure for the recommendations column, showing the dataframe was very inefficient. It is also due to this reason that the sparks dataframe object cannot be saved as a csv file. The complete codes for this local python implementation can be seen in the `models.ipynb` on the GitHub repository.

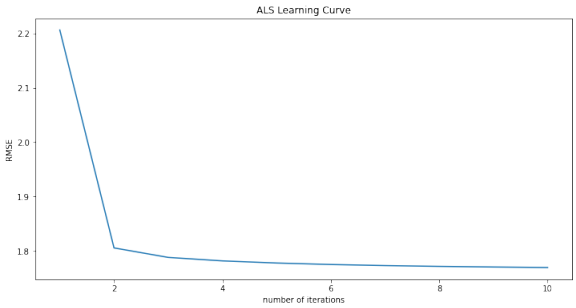


Figure 1: The learning curve for local python execution.

After the python implementation, we converted the codes to two `.py` files, `train_spark.py` which trains and outputs the best ALS model, and `recomm_spark.py` which predicts and recommends the top 500 books using the best ALS mode. These two files were then executed locally using pyspark instead of python. The run time for parameter tuning reduced to just 1743.17 seconds, a 56.92% increase in efficiency. The local pyspark implementation also returned the same model and prediction results as the python implementation, and it still wasn't able to show the recommendation dataframe.

After ensuring that the local pyspark implementation is successful, the datasets and `.py` files were uploaded to `hdfs:/user/cc6580/1_perc_csv/`. The same execution was then implemented using the dumbo cluster. The total time for parameter tuning on the cluster took 4109.25 seconds, which is not really a decrease from the local implementation. However, this might be due to the cluster being crowded when implementation took place. The hadoop implementation also returned `reg=0.2` and `rank=20` as the best model parameter values. However, it returned slightly different RMSE values. The complete tuning outputs can be seen in figure 6. Since pyspark does not have any plotting functionality, results from the learning curve was outputted as a table shown in figure 7. The learning time also did not improve from the local execution, possibly due to the same reason as the tuning time. However, using the cluster, we were able to show the final recommendation table, seen in figure 2. Unfortunately, since we were unable to process the full dataset, we were unable to test our implementations on the full dataset.

```
Showing the first 10 user recommendations
+-------+--------------------+
|user_id|     recommendations|
+-------+--------------------+
|   1829|[[810441, 7.70547...|
|  20924|[[706380, 6.01725...|
|  33412|[[488709, 6.36618...|
|  97413|[[2080484, 8.7668...|
| 102798|[[1775549, 3.2158...|
| 106535|[[273277, 8.72099...|
| 108560|[[573967, 3.23661...|
| 115602|[[504694, 6.49319...|
| 163046|[[448461, 6.81794...|
| 171723|[[115369, 7.65408...|
+-------+--------------------+
only showing top 10 rows

Total Runtime for recommendation: 1159.22 seconds
```

Figure 2: The first 10 rows of recommendation outputs for execution on hadoop.

5 LightFM

For our project, we chose to implement the LightFM model as another way of building the recommendation system. In order to train the LightFM model, we first had to convert our datasets into sparse matrix format, and ensure that the matrix for all train, validation, and test sets are of the same dimension, $number_{users} \times number_{items}$. Thus, we first created 3 copies of our dataset, one for train, one for validation, and one for test, and set the ratings for those data sets as 0. These 3 data sets will be used as the fill data sets later. Then, we created the 3 model data sets using a concatenation of the true and the fill data sets according to ensure that all of train, validation, and test includes all possible users and items. Then, we tried to convert these 3 data frames into COO sparse matrix representation.

5.1 Attempt 1: pandas pivot

The most straight forward solution to our problem is to pivot the datarames setting the unique `user_id's` as index and the unique `book_id's` as columns. Then, the values of the pivoted table can be extracted as a dense matrix, which can then be converted to a sparse matrix. However, since our data frame is very large, we were unable to pivot the table, receiving errors such as "Unstacked DataFrame is too big, causing int32 overflow".

5.2 Attempt 2: get dummies

Since we cannot pivot our dataframes, we tried to get dummy variables from our `book_id` column to try and imitate a pivoted dataframe. Although there were no errors, we waited for 2 hours for this action before finally giving up and interrupting the code. We have too many unique `book_id's` to extract dummies from.

5.3 Attempt 3: pandas groupby

We then tried to group our dataframe by both the `user_id` and `book_id` columns. This group by was implemented very fast and successfully. We then attempted to loop through each group to try and extract the id's as indices and ratings as values for our sparse matrix. However, once again, the for-loop was too inefficient to be completed.

5.4 Attempt 4: sparks pivot

Since pandas is notorious for being inefficient for large data sets, we tried to utilize spark for our goal. We created 3 spark dataframe objects from our pandas dataframe objects rather quickly. We then tried to pivot our sparks dataframe by using a combination of `groupby`, `pivot`, and `agg`. Unfortunately, this attempt also took way too long, and we eventually interrupted it.

5.5 Implementation on small data set

Since the down-sampled data was still too large for us to convert, we further down-sampled it to contain only 2000 instances with 7 unique `user_id's`. This time, we were able to use the methods described in our Attempt 1 to convert our 3 dataframes into COO format. Then, we trained a LightFM model using `warp` as our loss function. Then, we recommended 5 books to all the users in our data set. For illustration, results for 2 of the users can be seen in figure 8. Since we were unable to produce recommendations for

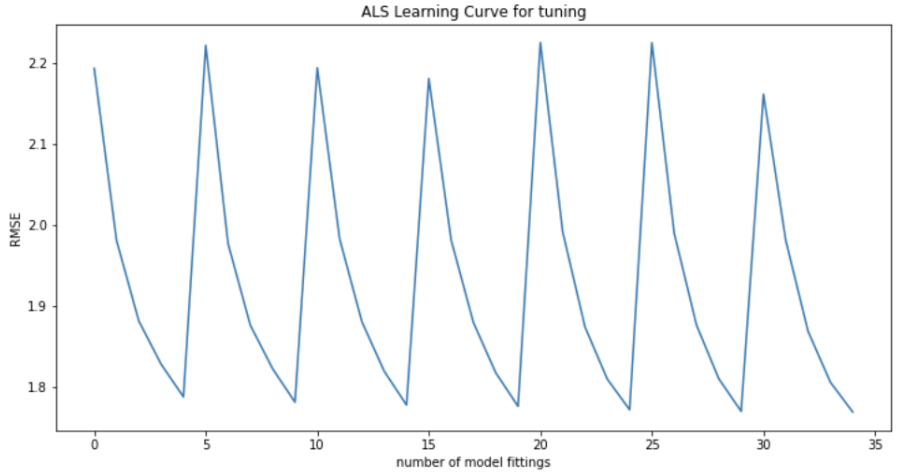
6 Conclusion

In terms of the failure of data splitting on full dataset, we think we should have missed some other solutions over this problem although we have tried almost every techniques learnt in class. We believe using sql in pyspark could have brought us to success, but the result continuously failed us for some possible dumbo reasons. In the future, we would pay more attention to spark dataframe module and methods to figure if any of them could enable the data splitting process to complete on dumbo. We are also willing to repeat current methods later after semester finishes when dumbo is not that busy.

Appendix A ALS model

```
8 latent factors and regularization = 0.001: validation RMSE is 2.193343389284863
8 latent factors and regularization = 0.01: validation RMSE is 1.9811209952256954
8 latent factors and regularization = 0.05: validation RMSE is 1.8814114788497316
8 latent factors and regularization = 0.1: validation RMSE is 1.8280880483045037
8 latent factors and regularization = 0.2: validation RMSE is 1.7877510027053956
10 latent factors and regularization = 0.2: validation RMSE is 1.7811731425963415
12 latent factors and regularization = 0.2: validation RMSE is 1.7776799510140204
14 latent factors and regularization = 0.2: validation RMSE is 1.7760735153726381
16 latent factors and regularization = 0.2: validation RMSE is 1.771874223015962
18 latent factors and regularization = 0.2: validation RMSE is 1.7698881990035666
20 latent factors and regularization = 0.2: validation RMSE is 1.7691436078132199

The best model has 20 latent factors and regularization = 0.2
```




```
Total Runtime: 4046.15 seconds
```

Figure 3: The parameter tuning outputs for local python execution.

```
+-------+-------+------+-----------+
|user_id|book_id|rating| prediction|
+-------+-------+------+-----------+
| 347669|    148|     0|   1.504346|
| 107119|    148|     4|  2.6149368|
| 186760|    148|     4|  2.1043317|
| 362708|    148|     5|   2.022663|
| 254718|    463|     0|0.026796421|
+-------+-------+------+-----------+
only showing top 5 rows
```

Figure 4: The first 5 rows for prediction output.

```
root
 |-- user_id: integer (nullable = false)
 |-- recommendations: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- book_id: integer (nullable = true)
 |    |    |-- rating: float (nullable = true)
```

Figure 5: The table schema for the recommendation dataframe.

```
**********************************************************************
8 latent factors and regularization = 0.001: validation RMSE is 2.232240115105491
8 latent factors and regularization = 0.01: validation RMSE is 1.9773831222597729
8 latent factors and regularization = 0.05: validation RMSE is 1.8774814175253718
8 latent factors and regularization = 0.1: validation RMSE is 1.8254337506268183
8 latent factors and regularization = 0.2: validation RMSE is 1.7873032284271886
10 latent factors and regularization = 0.2: validation RMSE is 1.783018820115905
12 latent factors and regularization = 0.2: validation RMSE is 1.7797219792074872
14 latent factors and regularization = 0.2: validation RMSE is 1.7746079848977405
16 latent factors and regularization = 0.2: validation RMSE is 1.7729467511417476
18 latent factors and regularization = 0.2: validation RMSE is 1.7711263530070416
20 latent factors and regularization = 0.2: validation RMSE is 1.767887271971055

The best model has 20 latent factors and regularization = 0.2
[model saved under '1_perc_csv/' as 'modelSaveOut'
[Total Runtime for tuning: 4109.25 seconds
```

Figure 6: The parameter tuning outputs for execution on hadoop.

```
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+
-----------------+-----------------+-----------------+-----------------+-----------------+
|   rmse|2.199132804019401|1.8069089330882329|1.7875243914475316|1.7802214817988473|1.7759458798858523|
1.7730697631863328|1.7711025117003099|1.7697138250483475|1.768677681867382|1.767887271971055|
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+
-----------------+-----------------+-----------------+-----------------+-----------------+
|maxIter|                1|                2|                3|                4|                5|
                6|                7|                8|                9|               10|
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+
-----------------+-----------------+-----------------+-----------------+-----------------+

Total Runtime for learning curve: 1342.51 seconds
```

Figure 7: The model learning outputs for execution on hadoop.

Appendix B LightFM

```
User 431147
Known book_ids:
438
443
536
Recommend book_ids:
7169
46407
7457
16317
1112
**********************************************************************
User 431161
Known book_ids:
459
1062
1065
Recommend book_ids:
52721
78539
860040
28581
1621
```

Figure 8: The example outputs for lightfm