DS-GA-1004 Big Data Recommender System

Peter Simone (ps4021)

May 11, 2020

1 Overview

Recommender Systems seem to impact nearly all decision making tasks nowadays. Whether we are searching for books, groceries, shows to stream, or apartments to rent, we are presented with curated options such as these almost daily. This report outlines my developing of a recommender system based off the Goodreads dataset [7][8][9], utilizing spark on the NYU dumbo cluster. The Goodreads dataset contains interactions between users and books, outlining the ratings (1-5 stars) that users gave as feedback about the books they have read, and also providing the books that they intend to read. In total, there are 228,648,342 interactions, and 876,145 unique users in the entire dataset. Given the dataset size, it is not one that can be reasonably processed on a single machine in its entirety, so this project works with distributed computing. The main objective of this project is to develop a recommender system to predict the top recommendations for a given user using Pyspark and the parallelized Alternating Least Squares (ALS) model [5]. I then explore a different type of recommender system, using LightFM [1] on my local machine, to compare its single machine efficiency to pyspark ALS as a function of input data size.

2 Data Processing

Given the large dataset, and the sometimes delicate processes on the dumbo cluster, I initially downsampled the data - based off users - to a more workable size, which I could work with locally for development purposes. Once the logic was in place, I transitioned to the dumbo hdfs cluster.

The following steps are ones that I generally applied to the data before training, or before comparing with LightFM, discussed in a later section:

- **Remove interactions where books had not been read yet (is_read == 0)** . This is because, these interactions have feedback listed as 0 stars. If I keep these interactions in the dataset, the model will try to learn this representation, which is completely misleading and will negatively impact the model.
- **Remove users where there were less than 10 interactions**. This, in combination with the previous step, cuts the size of the data pretty significantly, but removes users with little information provided. On the entire dataset, it trims total interactions to 111,662,303 and total unique users to 718,021.
- **Sort and repartition data**. These are shown to speed up computation. Partition values are something I could explore more, but I am repartitioning to the default 200 logical blocks.

The next step, for both cross-validation and training of the final model, is to split the data. This process was complicated by needing to split on user, rather than interaction, and by the fact that half the interactions per user in both the validation and testing splits had to be pulled back into the training set. I wrote custom functions to split appropriately, and wrote custom functions to cross-validate the model on all data in order to find a somewhat more curated set of hyper-parameters to use moving forward. In all, the training/validation/testing sets would follow a 60/20/20 split. To do this, I first split the dataset into training+validation users and then testing users. I created 3 separate dataframes consisting of the following information, which I checkpointed since the following dataframes would have actions performed on them many times. These checkpointed dataframes were as follows:

- **Dataframe (1):** Training and Validation interactions
- **Dataframe (2):** Testing interactions to be pulled back into Training
- **Dataframe (3):** Testing interactions to be kept for testing

For each fold of the cross-validation, dataframe (1) is split randomly by user and half the resulting validation interactions per user are pulled back into the training split. Windows functions and SQL CTE's make this quite easy. Then, dataframe (2) is added to the newly compiled training set. These dataframes are persisted through the session since they will be acted on many times in cross-validation. I created custom modules to handle all the pre-processing and cross-validation, of which I discuss further in the next section.

3 Model and Experiments

For modeling the data, the Alternating Least Squares (ALS) model is used [9]. This collaborative filtering model seeks to factorize a utility matrix in into user and item matrices, in a parallel manner. In ALS, while training the model, the user values are held constant while items are learned, then items are held constant while users are learned.

3.1 Hyper-Parameter Tuning

With the splits created in pre-processing, the ALS model is then trained for each combination of hyper-parameters. I chose a more naive evaluation approach for hyper-parameter tuning, using RMSE as my evaluation metric. This will only be the metric that I use to evaluate hyper-parameter tuning, but I will discuss other evaluation metrics of the final model later in the report. In RMSE, the order of predictions is not taken into account, but instead it will measure overall similarity, with higher penalties on those values differing more from the truth. The results are included in Table 1. "Rank" denotes the dimensionality of the latent factor representations, "regParam" is the regularization parameter, and "maxIter" is the number of cycles that the ALS algorithm is allowed to cycled through. All can be viewed as ways to account for overfitting, but there are computational tradeoffs for them, at least for "maxIter" and "rank".

rank	**regParam**	**maxIter**	**mean RMSE**	**std RMSE**
10	0.01	10	1.3906	0.0020
10	0.1	10	1.1994	0.0004
10	0.5	10	1.3026	0.0001
20	0.01	10	1.4455	0.0010
20	0.1	10	1.1906	0.0001
20	0.5	10	1.3030	0.0001
50	0.01	10	1.4694	0.0003
50	0.1	10	1.1785	0.0001
50	0.5	10	1.3029	0.0001

Table 1: 5-Fold Cross-Validation Evaluation scores of ALS model on all (pre-processed) data.

I did not report adjusted maxIter because increasing this value gave me all sorts of overflow errors. I did not report rank values higher than 50, because I would often be thrown memory issues or just have incredibly slow processes, even after allocating more memory to the spark session. I am likely sacrificing some performance by choosing this subset of hyperparameters, but it runs much more quickly. From this 5-fold cross-validation, the hyper-parameters of rank=50, regParam=0.1, and maxIter=10 lead to the lowest RMSE, and will be what I use moving forward in testing on the entire dataset. Results are very low variance, as given by the low standard deviations. Next steps were to apply this working logic, and use the hyperparameters I determined, to train a final model on the entire dataset. The code for above is all in my github repository, but examples of the command line executable scripts are in the Appendix.

3.2 Training/Evaluating the Final Model

Now I had working logic and a set of tuned hyper-parameters based off RMSE evaluation during cross-validation on all data. I could move forward with training a final model on all the available data. The steps taken to train the final model are :

1. take all the data, pre-process as outlined above
2. create a 60/20/20 split based off users
3. pull half of holdout set interactions per user back into training set
4. combine my training and validation sets
5. train the final model on this combined training/validation set

My "validation" evaluation was done through cross-validation as shown in the previous section, while below I evaluate the "out-of-sample" test set performance on the final trained model. Ideally, a more appropriate evaluation metric than RMSE would be used, since we are ranking books. Books typically take a long time to read, so it is important to put a large emphasis on those books ranked highest. I'll still record RMSE, but I will more importantly include MAP for the top 500 recommended books per user [4][5][6] below. Mean Average Precision (MAP) takes the mean of the average precision for each user, where average precision is the calculated precision for each position rank of interest up to a point, and averaged.[5] So if the highest predicted ranks are wrong, those values will persist through to negatively impact the MAP. To calculate this, I deem books with known ratings >= 3 as "relevant" books, which should ideally appear in the high end of predicted rankings. If the model predicts books that aren't relevant, meaning true rating < 3, then it will be penalized accordingly based on its predicted rank. I will not use *Precision at K* since order within the top 500 is not taken into account, and as mentioned, books take a long time to consume and there should be a higher penalty if the top recommendations are incorrect. Table 2 outlines the results for out-of-sample evaluation.

Evaluation Metric	**# of Recommendations**	**Value**
RMSE	ALL RECS	1.1747
RMSE	500	1.1803
MAP	500	0.9156

Table 2: Out-of-sample evaluation metrics for top 500 predicted recommendations.

I'll go into slightly more detail about how I calculate the evaluation metrics. I first transform the test set based off the final model. So this will create predictions *only* for those interactions that have "truth" in the holdout set, meaning only for those books that have been rated by a user. For RMSE for ALL RECS, I calculate RMSE for all data from the transformed data. For RMSE for top 500 recommendations, I use a Windows function to rank the transformed predictions per user, take only the top 500 ranks per user, and calculate RMSE based off those. MAP, the more valuable metric here, is a more intricate calculation. I create a list of relevant books per user, defined as the "truth" rating being >= 3. Then I use a Windows function to get the top 500 predicted books per user, order the set based off predicted rating, build an RDD joining the predicted top 500 ranked books with the set of relevant books for that given user, then compute MAP by referencing the relevant books.

The model seems to perform very well in predicting the top 500 book recommendations. Generally, the predicted feedback is similar to the actual feedback according to RMSE, and more importantly, the rank is predicted well according to MAP. A caveat is that very few users actually read 500 books. We could definitely recommend 500 books to a user, but when it comes to evaluating the model given the requirement of evaluating on 500 items, I believe this could lead to discrepancies in evaluation, specifically for MAP, since I'm not entirely sure how pyspark handles these cases where number of rated books is less than 500. Code to develop the above information is given in the Appendix, and can be found on my github repository.

4 Extension

To experiment with the efficiency and performance of distributed computing and pyspark's parallel ALS implementation, I'll compare the performance of ALS on the dumbo cluster to the performance of LightFM [1][3][2] on a single machine, in terms of both MAP and speed (seconds). I chose MAP for the same reason I mentioned in the previous section, that it serves as a great evaluation metric for ranking books. I'll evaluate this as a function of dataset size as well, scaling up to see at what point parallel ALS may outperform the single machine LightFM.

LightFM can be used to model implicit or explicit feedback mechanisms, and is a hybrid of collaborative filtering and content-based modeling [1]. It is used to overcome the cold-start issue, but can also be used purely as a collaborative filtering method [2]. Latent factors are defined as functions of embeddings of the feature representations, and the objective is to maximize the likelihood conditional on feature parameters, and is optimized with asynchronous SGD [2]. For now, I use it solely for collaborative filtering of explicit feedback. For the purpose of this extension, I will not be experimenting with optimizing hyperparameters for either model. I'll simply use the model default values for both ALS and LightFM, aside for number of epochs in LightFM, which I upped to 10 to match the default ALS "maxIter" value of 10. On a given subset of the data, I'll run the model 5 times to gather both run time and evaluation metrics. For evaluation at each downsampled iteration, I'll split data 80/20, pull half of the testing interactions per user back into training, and evaluate the out-of-sample data. Again, I'm using mostly default hyper-parameters, meaning I am not attempting to optimize either model, so I think the timing metric will be more valuable than the performance metric.

LightFM does not have a built in MAP evaluation metric. I wrote a custom function to handle this. But briefly, I predict ratings only for interactions where there a book has been read in the test set. Relevant books are defined as books rated >= 3. For each user, I calculate precision for top 50 predicted books, and if there are less than 50 actual books, it predicts $min(k, books\_read)$. MAP is the average precision across all users.

The first step was that I decided I was only going to evaluate up to 20% of the original dataset. This is mainly due to the memory constraints I have on my local machine. I pre-processed the data as mentioned in Section 2, downsampled to 20% of users, then wrote out a parquet file (for pyspark ALS implementation) and a csv file (I transfer this over to my local computer via Github push/pull for LightFM implementation, and read the chunks of data in iteratively). I made sure to isolate the timing metrics that I show below to be of training and predicting/evaluating only, so not including any pre-processing steps that take place in between. I use the mean value for time metrics, but I'll note that the first iteration of training the model on hdfs was significantly slower than the following iterations. I did not look into the reason why. I ended up running much smaller subsets of data than I originally planned for LightFM, since I found that once going above only 5% of users, the LightFM evaluation step specifically was taking over an hour (given that I am running 5 iterations at each downsample, this adds up to be quite timely). Results are plotted below, where evaluations are performed on top 50 recommended books, which is an arbitrary value that I chose.

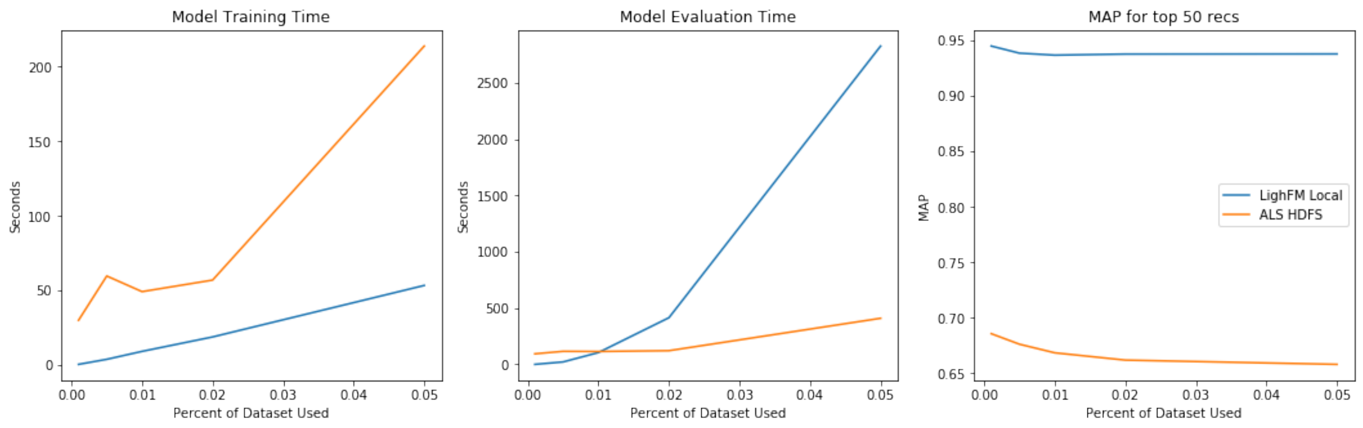

Figure 1: Comparison of LightFM single machine recommender system to Pyspark ALS parallelized recommender system.

It is clear that the LightFM implementation significantly outperforms the ALS implementation for MAP, although I'm sure these are nowhere near optimized models, and I'd pay less attention to this metric. Also, MAP for LightFM is not a built in function. Model training times scale similarly, with LightFM training more quickly in the provided ranges. Where ALS significantly outperforms LightFM is on the model evaluation time, where the LightFM time blows up, while ALS stays relatively unchanged in these ranges. ALS on HDFS would be able to scale much better than LightFM on a single machine as I continue to increase these values, which I believe is expected given how much data there is. To back this point, the plot below shows the times for model evaluation with more data on ALS, the discrepancy in time spent is clear, even if I didn't run LightFM with more data.

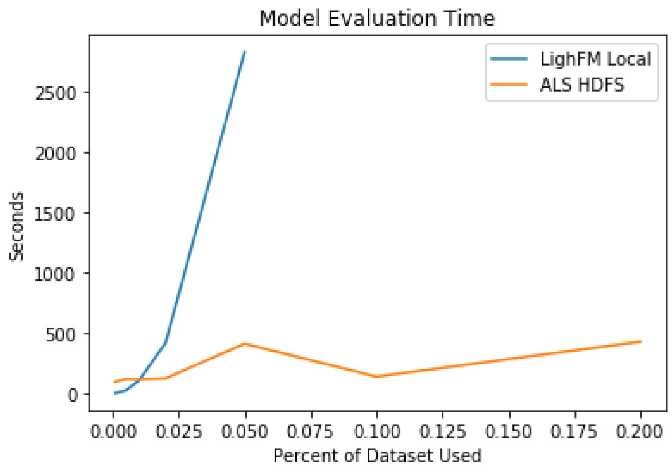

Figure 2: Comparison of LightFM to ALS for model evaluation time only. LightFM was not reported with the higher range of data, due to processing time. It is clear HDFS leads to relatively consistent time to evaluate the model with more data, compared to LightFM.

To generate the data above, custom scripts can be executed as given in the Appendix. For pyspark ALS, the spark-submit script is executable from the dumbo cluster, and for LightFM, the python script is executable on my local machine (local python LightFM). I think that LightFM is definitely a great model to use if there is less data, or more available resources. Given the quantity of data, the ALS model on HDFS seems to more efficiently output results.

5 Appendix : Code

LightFM library is required for this project

```
conda install -c conda-forge lightfm
```

Examples of custom command line executable scripts.

To cross-validate the model

```
spark-submit spark_submission.py --CV_file_out=Cross_Validation.csv \
--rank 10 20 50 regParam 0.5,0.1,0.05
```

Also to cross-validate the model, but assuming you want to reprocess/split the data into training+validation / testing sets and then move forward with cross validation

```
spark-submit spark_submission.py --CV_file_out=Cross_Validation.csv \
--Recreate_Split=True --rank 10 20 50 regParam 0.5,0.1,0.0 \
--interactions_file=[path to full interactions file on hdfs] \
--users_file=[path_to_full_users_file_on_hdfs] \
--books_file=[path_to_full_books_file_on_hdfs] \
--percent_downsample=0.2
```

Snippet of code for pre-processing all data, training the final model, and evaluating performance.

```
>>> import data_preparation
>>> data_prep = data_preparation.Prep_Data(spark)
>>> interactions = data_prep.Get_Data_HDFS([path_to_interactions_data],\
    [path_to_users_data],[path_to_books_data])
>>> interactions = data_prep.Trim_LowNum(interactions,min_allowed=10,cut_not_read=True)
>>> train_val,test_to_train = data_prep.Create_TestSet(interactions,percent_train=.6)
>>> spark-submit spark_submission_TrainFinalModel.py --Model_File_write=[path_to_write_model] \
    --rank=50 --regParam=0.1
>>> spark-submit spark_ModelPerformance.py \
    --Model_File=[path_to_model] \
    --path_assess=[path_to_holdout_set] \
    --num_recommend=500
```

For the extension, this is executable from the dumbo cluster

```
spark-submit spark_extension.py --Data_File='UseForExtensionHDFS_20percentDownsample.parquet' \
--Original_Downsample=0.2 --Downsampled_percents .001 .005 .01 .02 .05 .1 .2 --num_run=5 \
--precision_at_k=50
```

Also for the extension, this is executable on my local machine (local python LightFM)

```
python local_extension.py --Data_Path=extension/UseForExtensionLocal_20percentDownsample.csv/
--Original_Downsample=0.2 --Downsampled_percents .001 .005 .01 .02 .05 --num_run=5 --precision_at_k=50
```

References

[1] Maciej Kula. Metadata embeddings for user and item cold-start recommendations. In Toine Bogers and Marijn Koolen, editors, *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.*, volume 1448 of *CEUR Workshop Proceedings*, pages 14–21. CEUR-WS.org, 2015.

[2] Maciej Kula. Metadata embeddings for user and item cold-start recommendations. *CoRR*, abs/1507.08439, 2015.

[3] Nasir Safdari. If you can't measure it, you can't improve it !!! NOV 2018. Accessed on 2020-05-05.

[4] Apache Spark. Evaluation metrics - rdd-based api. https://spark.apache.org/docs/2.3.0/mllib-evaluation-metrics.html. Accessed on 2020-05-05.

[5] Apache Spark. pyspark.mllib package. https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.evaluation.RankingMetrics. Accessed on 2020-05-05.

[6] Apache Spark. Spark ml cookbook (python). MAY 2017. Accessed on 2020-05-05.

[7] UCSD. Goodreads datasets. https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home. Accessed on 2020-05-05.

[8] Mengting Wan and Julian J. McAuley. Item recommendation on monotonic behavior chains. In Sole Pera, Michael D. Ekstrand, Xavier Amatriain, and John O'Donovan, editors, *Proceedings of the 12th ACM Conference on Recommender Systems, RecSys 2018, Vancouver, BC, Canada, October 2-7, 2018*, pages 86–94. ACM, 2018.

[9] Mengting Wan, Rishabh Misra, Ndapa Nakashole, and Julian J. McAuley. Fine-grained spoiler detection from large-scale review corpora. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2605–2610. Association for Computational Linguistics, 2019.