DSGA 1004 Big Data Project - Group 45

ELLIOT LEE (EL3418) and HARSHANAND B A (HB2474)*, NYU Center For Data Science, USA

Repository: https://github.com/nyu-big-data/final-project-group_45

ACM Reference Format:
Elliot Lee (el3418) and Harshanand B A (hb2474). 2022. DSGA 1004 Big Data Project - Group 45. 1, 1 (May 2022), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 TRAIN/VALIDATION/TEST SPLIT [NEW_SPLIT_SCRIPT.PY]

Author: Elliot

Analysis: Elliot

The initial train/validation/test splits, located in split_script.py, were done through Spark. The script worked by getting a list of the unique userId values, then splitting them up. Roughly, the first 2/3 of the userIds were assigned the value 'train', 1/6 were assigned 'validation', and 1/6 were assigned 'test'. Then, samples of the dataframe were taken, with varying rates according to the value assigned to that particular userId. 100% of rows with value 'train' were sampled, 30% of rows with value 'validation' were sampled, and 30% from rows with value 'test'. These sampled values would represent the training set, then the remaining rows would be calculated via differencing and would go into their respective validation or test sets. The goal when writing this script was to emulate Professor Brian's suggestion on Brightspace.

However, the script did not actually perform as intended, perhaps due to differences between Spark and local implementation. The expected behavior was that the first 400 unique userIds would appear in train (as it had a sampling rate of 1 for the first 2/3 userIds), however many userIds were missing when inspected. This was further confirmed in the checkpoint feedback. Therefore, a new script needed to be written (stored in new_split_script.py). This script uses Pandas instead, and the implementation is a lot simpler. For each userId, 25% of the initial interactions are sampled and put into the validation set. Then, the dataframes are differenced to obtain the remaining interactions and another 25% are sampled from the remaining interactions into the test set, then the rest of the interactions for into the training set. This produces data splits that are disjoint, and every dataset contains at least some interactions from every user.

2 EVALUATION CRITERIA

The main evaluation criteria, as suggested, is MAP@100. We also evaluate accuracy for the popularity baseline, and RMSE for the ALS model.

Authors' address: Elliot Lee (el3418), el3418@nyu.edu; Harshanand B A (hb2474), hb2474@nyu.edu, NYU Center For Data Science, 60 5th Ave, New York, NY, USA, 10011.

3 POPULARITY BASELINE [POPULARITY_MODEL.PY]

Author: Harshanand

Analysis: Harshanand

The objective of the popularity baseline model is to calculate top n (in this case 100) movies based on some popularity metric. The very naive approach would be to calculate average rating of each movie in the training set and then rank them in descending order. However there are few problems with such an approach. What if a particular movie was rated by very few people ? In this case taking the average would not be an ideal solution. Hence to solve this problem we took a "weighted score" by factoring in the number of ratings given for a particular movie. In such a setting movies with high score would essentially have a high average rating and sufficient number of ratings. This eliminates the inconsistency that originates from just taking the average rating alone.

3.1 Results

3.1.1 Small Dataset.

Validation accuracy: 0.5312599427298759
Test accuracy: 0.6236482753177046
Validation mAP: 0.0006
Test mAP: 0.0007

3.1.2 Large Dataset.

Validation Acc:0.9225328552156452
Val. mAP:0.00019517938110874528
Test Acc:0.9224456174181017
Test mAP:0.00019516552455690092

4 ALS (LATENT FACTOR MODEL) [ALS.PY]

Author: Elliot

Analysis: Elliot, Harshanand

For the ALS model, we set nonnegative=True, because we are observing ratings with only positive value. We also set implicitPrefs=False, because we are observing explicit ratings, not implicit ones. Finally, we set coldStartStrategy='drop' to ensure we do not get NaN values. Next, we use Spark's CrossValidator on the small dataset in order to tune our hyperparameters. The best results we obtained were with rank=100 and regParam=0.15. Regarding maxIter, we were originally able to fit and calculate precision for maxIter=3. However, after we fixed the errors in our script to produce the Train/Val/Test splits and obtained new datasets, the code struggled. It was able to fit the model and produce recommendations, but the error was in how the results were transformed afterwards in order to feed into Spark's RankingMetrics evaluator. In order to do so, the sc.parallelize() function was used, which turned out to be a mistake, as the function works fine on small datasets but was not designed to scale to incredibly large sets, and create memory issues. Using the new datasets and sc.parallelize(), the code would only run with maxIter=1 and produced a MAP@100 score of 3.31E-7. Therefore, in order to obtain better results, the code for the ALS model was re-written to use a custom MAP function, lifted from the code for the Popularity Baseline, instead of using sc.parallelize() to utilize Spark's RankingMetrics. This lead to good improvements in the MAP@100 (1.03E-4 using the same rank and regParam, but

with maxIter=3). This also lead to performance speed ups - running the script on the faulty dataset with less data than intended still took around 1.5 hours to complete, but this script was able to finish on a larger dataset in around 30 minutes. The new code also allowed us to experiment with a suggestion from the checkpoint feedback to increase the number of iterations, but doing so did not lead to any noticeable increases to speed or accuracy when using max_iter=5.

Comparing the ALS model on the small and large datasets, the biggest noticeable difference was in the time to fit the model: on the small dataset, it took ~0.0087 seconds, while on the large dataset it took ~0.1043 seconds. While only fractions of a second, it is still an increase of about 12 times. Curiously, the time to transform the model and give recommendations did not get worse - on the small dataset it took ~0.003 seconds, compared to about ~0.002 seconds on the large dataset. When observing RMSE values, the accuracy seems to be the same (both around 0.9). However, it took almost twice as long to calculate RMSE on the large dataset (~0.003 seconds vs ~0.006 seconds). Finally, the MAP scores of the model trained on the large dataset are better (~1E-4 vs ~1E-5). This is expected, as more data should theoretically help give better recommendations and avoid overfitting.

4.1 Parameter Config

regParam: 0.15
rank: 100
max_iter: 3

4.2 Results

4.2.1 Small Dataset.

Time to fit model: 0.008704197999999996 seconds
Time to transform and give recommendations: 0.0032634609999999675 seconds
Time to calculate RMSE: 0.0036570899999999185 seconds
RMSE: 0.904395548727064
mAP@100: 1.565149480562146E-5

4.2.2 Large Dataset.

Time to fit model: 0.10425484400000007 seconds
Time to transform and give recommendations: 0.0018419579999999769 seconds
Time to calculate RMSE: 0.005929198999999996 seconds
RMSE: 0.8851815970130998
MAP@100: 0.00010285021517227581

5 SINGLE MACHINE IMPLEMENTATION (EXTENSION) [EXTENSION.PY]

Author: Elliot
Analysis: Elliot, Harshanand

For this extension, we created a model in LightFM. To do this, we had to convert our dataframes into sparse matrices of interactions. We map the user IDs and movie IDs to indices, feed them into a COO matrix, then convert to CSR. We start by doing this for the ratings dataframe that contains all of the interactions. However, we cannot feed our

train/validation/test dataframes into the same function, because we need to have sparse matrices that are all of the same dimension, and each dataset is missing interactions. To resolve this issue, we take an alternative approach. We create a copy of the ratings sparse matrix, compare each dataset to the ratings dataset, then 0 out the interactions which exist only in the ratings dataset. This allows us to obtain a sparse matrix that contains all of the correct interactions and is also of the correct shape. We then run some assert statements to confirm each matrix is disjoint from each other, then feed the train matrix to the model for fitting. Testing on epochs from 1 to 10, we notice improvements in MAP scores. Training MAP goes from around 0.2 to 0.3, and Validation/Test MAP scores go from around 0.01 to 0.05. For this model, we use loss='warp'. This loss function optimizes for precision@k, according to *WSABIE: Scaling Up To Large Vocabulary Image Annotation* by Weston et al., the paper in which WARP was first introduced. Because of this, we should expect MAP to be better in LightFM. Indeed, we see that this is true - in general, LightFM took much longer to execute (as it is a single machine implementation), but gave better MAP@100 scores.

On the small dataset, ALS took just under 0.01 seconds to fit, while LightFM took a little over 0.3 seconds. On the large dataset, ALS took 0.1 seconds to fit, while LightFM took around 145 seconds. Here we can see the efficiency of cluster computing - ALS was able to fit the large dataset magnitudes times faster than it took LightFM to fit the small dataset. An even bigger bottleneck, though, was the time it took to calculate the MAP scores. While it only took fractions of a second to calculate precision on the small dataset for LightFM, it took hundreds of seconds to calculate precision for the validation and test set, and over 1000 seconds for precision on the training set. Compared to the ALs model, which took seconds to calculate RMSE and MAP, regardless of the size, and it is clear the single machine implementation is nowhere near as efficient.

The benefit to the LightFM model, however, is in it's loss function. For both the small and large datasets, the MAP@100 was ~0.05. This is much better than the MAP@100 for the ALS model, which had MAP scores closer to 1E-4 and 1E-5.

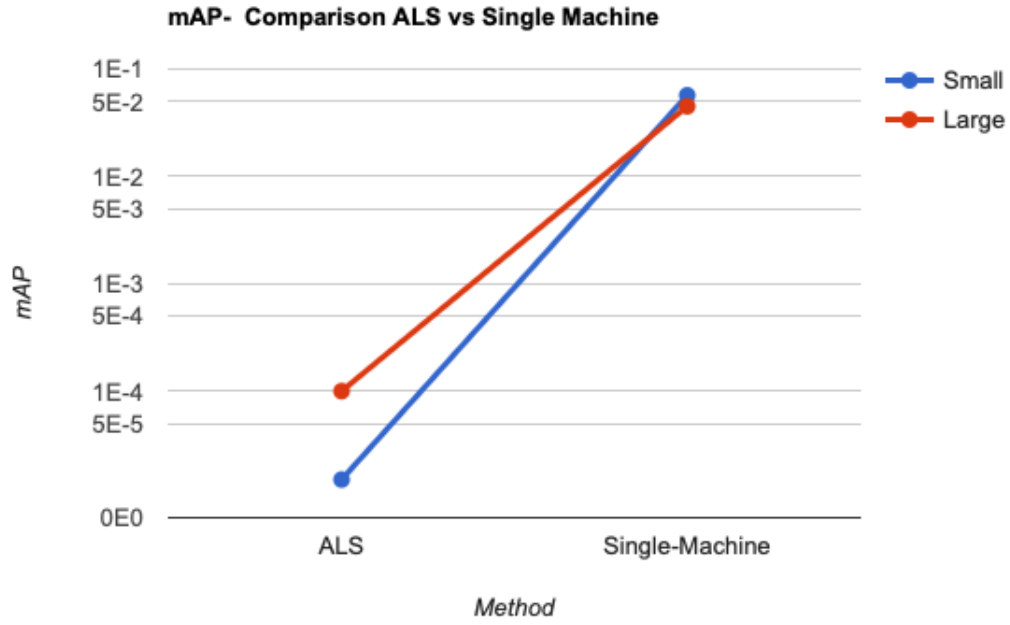
5.1 Results

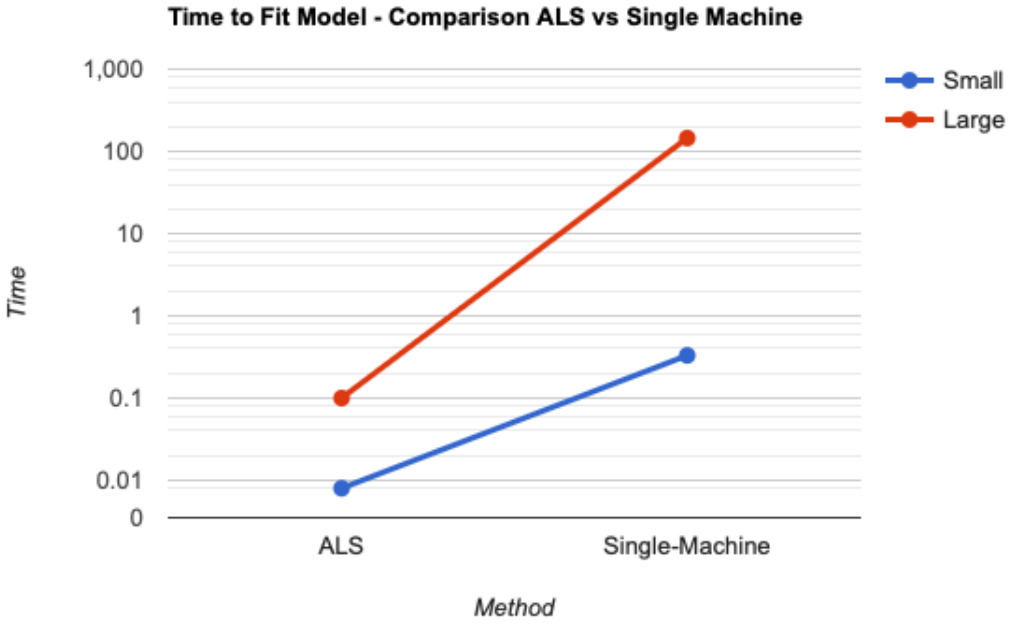
5.1.1 Small Dataset.

Time to fit model: 0.33740075199999886 seconds
Time to calculate training precision: 0.7290513840000017 seconds
Time to calculate validation precision: 0.46675056299999973 seconds
Time to calculate test precision: 0.39438100100000106 seconds
Training MAP: 0.20457377
Validation MAP: 0.07532787
Test MAP: 0.05703279

5.1.2 Large Dataset.

Time to fit model: 145.122320273 seconds
Time to calculate training precision: 1037.019329016 seconds
Time to calculate validation precision: 672.8502805200001 seconds
Time to calculate test precision: 601.777096153 seconds
Training MAP: 0.13246545
Validation MAP: 0.059357993
Test MAP: 0.04508753


mAP- Comparison ALS vs Single Machine
1E-1
5E-2
1E-2
5E-3
1E-3
5E-4
1E-4
5E-5
0E0
mAP
ALS
Single-Machine
Method
Small
Large


Time to Fit Model - Comparison ALS vs Single Machine
1,000
100
10
1
0.1
0.01
0
Time
ALS
Single-Machine
Method
Small
Large