Collaborative-Filtering Based Recommender System

DS-GA 1004 Final Project

Alex Herron
Center for Data Science
New York University
New York, New York
ah5865@nyu.edu

Dhruv Saxena
Center for Data Science
New York University
New York, New York
ds6802@nyu.edu

Sukrit Rao
Center for Data Science
New York University
New York, New York
str8775@nyu.edu

DATA PARTITIONING

To split the data into train/test/validation sets, we first split 70% of the total observations into the training set. Specifically, we included 70% of the data from each user, in order to guarantee that there would be sufficient training observations for the recommender system to learn each user's preferences. Without ensuring that each user's ratings appeared in the training set, the model would need to randomly evaluate users whose ratings were excluded from the training set. Next, we split the remaining 30% of total observations into validation and test sets, which were split by user. For this, we randomly split all the users into two groups (let's say group A and group B), then split those groups of users into validation and test sets. Because different users reviewed a different number of movies, the 30% is not perfectly split into 15% validation and 15% test. For example, users in group A might review, on average, more movies than users in group B. However, since the datasets are so large, the validation and test splits end up being generally close to 15% of the total observations.

POPULARITY BASELINE MODEL

For the baseline model, we calculated average ratings for each movie, with an additional damping factor, beta, to account for users with low number of ratings. These averaged ratings served as the predicted rating for each given user. Moving forwards, this serves as a useful measure, since we can now compare a collaborative-filter based model to this baseline.For the baseline model, we calculated average ratings for each movie, with an additional damping factor, beta, to account for users with low number of ratings. These averaged ratings served as the predicted rating for each given user. Moving forwards, this serves as a useful measure, since we can now compare a collaborative-filter based model to this baseline.

ALTERNATING LEAST SQUARES MODEL

For the untuned Alternating Least Squares (ALS) model, we fit our model on the training set for 10 epochs using a regularization parameter of 0.01 and a latent factor rank of 10. This setting was used for both the small and large datasets. For the final model, we trained for 15 epochs using the rank and regularization parameter values obtained from hyperparameter tuning.

EVALUATION

In this section, we describe the metrics that we use throughout our experiments.:

1. Precision at k: It is the average precision of all queries, truncated at a particular ranking position k. In all our experiments we use k = 100
2. Mean Average Precision: It is defined as the mean average precision of the first k rankings of all queries. In all our experiments we use k = 100
3. Normalized Discounted Cumulative Gain (NDCG): It is a measure of the relevance or quality of the set of prediction results. It is calculated for all queries and is truncated at a particular ranking position k. In all our experiments we use k = 100

HYPERPARAMETER TUNING

For the baseline model, we determined the optimal beta value by optimizing over the normalized discounted cumulative gain (NDCG) on the validation dataset, since we observed that metric to be the most sensitive to changes in beta. For the small and large datasets, we obtained the following results:

BASELINE RESULTS

Experiment	Dataset	Precision at 100	Mean Average Precision	NDCG
Untuned Baseline	Small Data	7.272e-2	3.779e-2	1.643e-1
	Large Data	2.521e-2	1.833e-2	7.694e-2
Tuned Baseline	Small Data	7.862e-2	4.406e-2	1.822e-1
	Large Data	5.078e-2	4.351e-2	1.541e-1

Table 1: **Validation metrics for baseline model**

Experiment	Dataset	Precision at 100	Mean Average Precision	NDCG
Untuned Baseline	Small Data	3.934e-4	1.431e-4	7.402e-4
	Large Data	1.503e-6	3.243e-8	1.537e-6
Tuned Baseline	Small Data	7.748e-2	5.489e-2	1.993e-1
	Large Data	5.084e-2	4.379e-2	1.547e-1

Table 2: **Test metrics for baseline model**

ALTERNATING LEAST SQUARES RESULTS

Experiment	Dataset	Precision at 100	Mean Average Precision	NDCG
Untuned ALS	Small Data	1.587e-2	1.451e-3	2.202e-2
	Large Data	3.663e-7	7.853e-6	1.142e-5
Tuned ALS	Small Data	2.292e-2	5.499e-3	4.982e-2
	Large Data	5.03e-2	1.81e-2	1.03e-1

Table 3: **Validation metrics for ALS model**

Experiment	Dataset	Precision at 100	Mean Average Precision	NDCG
Untuned ALS	Small Data	1.495e-3	1.452e-2	1.998e-2
	Large Data	5.692e-7	6.772e-6	9.819e-6
Tuned ALS	Small Data	2.879e-2	8.146e-3	6.332e-2
	Large Data	4.983e-1	1.872e-2	1.154e-1

Table 4: **Test metrics for ALS model**

EXTENSION 1: LIGHTFM

In this extension, we compared Spark's parallel ALS model to a LightFM model. We measured differences in precision@k and model fitting time, all as a function of the training set size. Overall, we used 10 different sizes of training data, using 10% increments from 10% to 100% (using train.sample(frac) to create such splits). Then, to prepare data for LightFM, we used the Dataset() feature of LightFM, and then fit the unique values of userId and movieId. Next, we built interactions using build_interactions([tuple(i) for I in .values]). We chose warp as the loss function, as it was recommended in the LightFM documentation for optimizing the top of a recommended list using precision@k. Once the model was loaded and fit, we calculated the precision@k, while also recording the time it took for each iteration of model fitting as a function of the training set size. This was run for both the validation and test datasets.

In conclusion, LightFM outperformed our best hyperparameter-tuned ALS model in terms of accuracy as well as efficiency. In terms of loading time, even on a local machine, most of the LightFM evaluation was spent on loading the large files of validation and test sets. The inference part took less than 5 mins on average, ranging from 3 to 8 minutes. The accuracy for LightFM increased gradually for the increment in training dataset. However, with more flexibility and logistic support, we can expect the ALS model to do better as the performance for ALS improved as we moved from small to large, but LightFM's performance ranged from 0.06 to 0.08 across both.

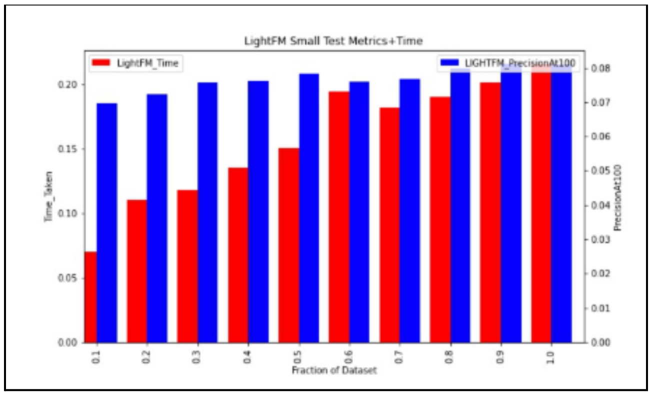

Figure 1: **LightFM Experiment Results on the Small Test Dataset**

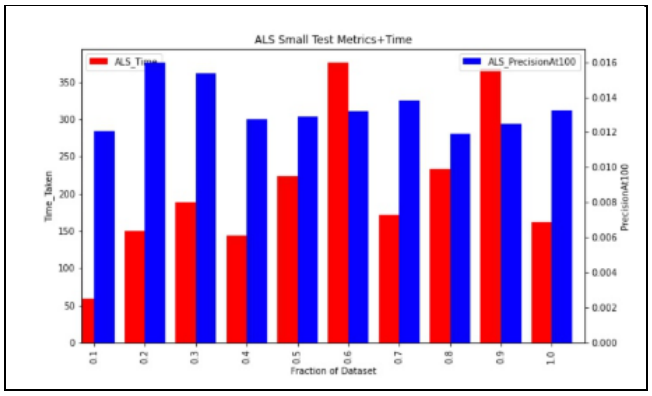

Figure 2: **ALS results on the small dataset**

Setup Instructions

We used the lightfm python package to perform the experiments for extension 1. It can be installed using the following command

```
$ pip install lightfm
```

EXTENSION 2: FAST SEARCH

In this extension, we benchmarked the performance of the Non-Metric Space Library (NMSLIB), an accelerated search library, against directly querying a Spark ALS model (brute-force method) to get the top movie predictions for a user. NMSLIB improves performance by constructing a Hierarchical Navigable Small World (HNSW) data structure, a graph data structure that finds the approximate nearest neighbors for a given user query in logarithmic time.

In our implementation, we used the following values of parameters to create the index

1. M - defines the maximum number of neighbors in the zero and above-zero layers
2. ef - increases the quality of the constructed graph at the expense of longer retrieval time
3. space - determines the metric that the algorithm uses to compute nearest neighbors

Based on the documentation, we use M = 20, ef = 200 and space = cosinesimil. Finally, the last step in the construction of the index involves adding the ItemFactors extracted from our trained ALS model to this index, upon which the graph is constructed. During test time, we performed batch inference for all users using the index.knnQueryBatch() method where we passed the userFactors for all users extracted from the ALS model. We passed in parameter k=100 to find the top 100 predictions for each user.
To benchmark the performance, we performed three separate runs on the small and large validation and test datasets. Here, we report the average time taken for the three runs.

Experiment	Val Small	Val Large	Test Small	Test Large
ANN-Avg Run Time	2.97	3.09	2.91	2.96
ALS-Avg Run Time	12.80	220.58	13.13	226.93

Table 5: **Benchmarking Results From Approximate Nearest Neighbor Algorithm vs ALS Model (Brute-Force method)**

From the table above, we conclude that while there is some additional overhead of creating the index and associated graph, the overall inference time using nmslib is significantly lower than using the brute-force method.

Setup Instructions

We used the nmslib python package to perform the experiments for extension 2. It can be installed using the following command

```
$ pip install --no-binary :all: nmslib
```

COLLABORATION STATEMENT

Credit to Alex Herron for executing training jobs for the ALS model, and for optimizing the ALS model training pipeline. Credit to Dhruv Saxena for developing the code for the evaluation metrics and resources pertaining to extension 1, and optimizing the ALS model training pipeline. Credit to Sukrit Rao for developing the code for the baseline model, the initial ALS model training pipeline, and resources pertaining to extension 2. All three members contributed to developing the code and logic for creating the data splits, performing hyperparameter tuning for the ALS model, and collating the results and writing up the report.

KEYWORDS

-Precision at k, mean average precision, normalized discounted cumulative gain, hierarchical navigable small world data structures,

ACKNOWLEDGMENTS

We would like to thank our Professor Brian McFee for his assistance and guidance through the development of this paper.

REFERENCES

[1] Leonid Boytsov and Bilegsaikhan Maidan. 2013. Engineering efficient and effective non-metric space library. In Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings, volume 8199 of Lecture Notes in Computer Science, pages 280– 293. Springer

[2] Yury A. Malkov and Dmitry A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. CoRR, abs/1603.09320.

[3] Maciej Kula. 2015. Metadata embeddings for user and item cold-start recommendations.In 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