Building Scalable Recommender Systems for Books :

with Apache Spark's Alternating Least Squares Module

Team Deep Coral Hyun Jung (hj1339@nyu.edu) Sujeong Cha (sjc433@nyu.edu)

1. Overview

This project aims to build a book recommender system using Apache Spark's machine learning library, MLlib.

|| Why Apache Spark?

Python, a traditional but popular tool, offers easy-to-use machine learning packages, but they are deployed on a single machine and consequently restricted in terms of scalability. On the other hand, Spark takes advantage of parallelism and can speed up ML performance on "Big Data". Hence, Spark MLlib is an appropriate choice for our book interactions dataset considering its size (4.32 GB in CSV).

|| Sneak-peek into Data

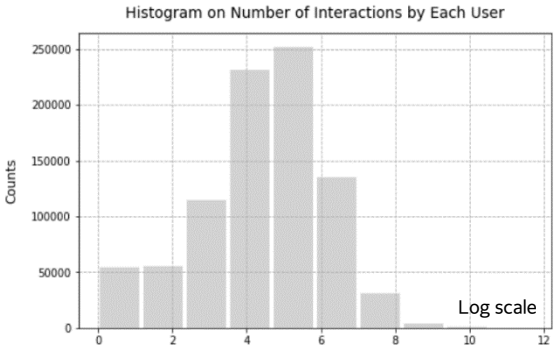

Figure 1. Histogram on Interactions by Each User

The provided Goodreads dataset contains histories of 876,145 users and 228,648,342 interactions in total. Each row represents a single interaction between a user and a book. The five columns of the dataset include *user_id(int), book_id(int), is_read(bit), rating(1-5 scale),* and *is_reviewed(bit),* where rating measures how much a user likes a corresponding book, which is considered as explicit feedback associated with direct user preference. As can be seen from ***Figure 1***, the number of interactions by each user is skewed to the right even in the log scale; the summary statistics show that the largest number of interactions a user has is 119,777 while 75% of users only have at most 265 interactions.

	userID	intByUser
min	0.000000	1.000000
25%	219036.000000	32.000000
50%	438072.000000	100.000000
75%	657108.000000	265.000000
max	876144.000000	119777.000000

Figure 2. Summary Statistics on Number of Interactions by Each User

|| Project Steps Summary

We started from a bare model and gradually enhanced its performance by the following step-by-step approach.

Steps	Data	Model (ALS)
1) Base Model	· 1% of users	· Default param
2) Filtered Data (Extension 1)	· is_read = 1 · interactions > 30 · 1% of users	· Default param
3) Param. Tuning	↑	· Rank Tuning · RegParam Tuning
4) **Larger Subset**	· is_read = 1 · interactions > 30 · 10% of users	· Optimal Rank, Reg
5) Fast Search (Extension 2)	↑	· Optimal Rank, Reg. · NMSLIB HNSW indexing

※ disclaimer: tried hyperparameter tuning on 100/50/30/20% subsamples as well but job aborted on Dumbo due to large size

2. Data Processing

|| Subsampling

Due to the limited computing power of the dumbo cluster, we randomly subsampled 1% of the total distinct users from the full dataset and then took all of their interactions to prototype the model.

|| Data Splitting

We constructed train, validation, and test splits of the subsampled and filtered data by selecting 60% of distinct users to form the training set, 20% to form the validation set, and the rest of 20% users to form the test set. For each validation and test user, half of their interactions were taken and added to the training set, and the other half was held out for validation and test. For the whole process, we used the fixed random seed(42) so the validation scores can be comparable across different versions of runs.

Data Filtering (Extension 1)

In order to improve the model performance, we adopted two modifications on the dataset, which includes discarding users with less than 30 interactions and dropping the instances where the rating feedback is not provided (*is_read* = 0) to examine their impacts on the baseline model performance.

|| Drop Unread Books

Since ratings from users that have not read given books are not useful, we dropped the data where *is_read* = 0 and only kept the data provided from users who actually have read the books. Moreover, we wished to preclude the possibility that a model misleadingly interprets zero-ratings (not-yet-rated) as negative feedbacks from users. Nearly 49% of the interactions (112,131,203 out of 228,648,342) belong to this category.

|| Removal of Users with < 30 Interactions

Upon analyzing data distribution, we found that about 31% of distinct users have interactions fewer than 30. Considering users with few interactions may not provide enough data for evaluation, we decided to discard the user with less than 30 interactions from our data as they would only take up volume in our processing memory and time. Since the removed interaction data only takes 3.03% of the total interactions, we expected the model performance would not be significantly affected by this action.

	Full Data	Training (60%)	Validation (20%)	Test (20%)
1% subsamp. (base model)	8,825	8,746	1,707	1,797
Filtered	564,906	564,906	112,609	113,226
1% subsamp. + filtered (Extension 1)	5,726	5,726	1,151	1,189

Figure 3. Unique User ID Count

3. Model and Experiments

|| Model: Alternating Least Squares (ALS)

Alternating Least Squares (ALS) matrix factorization decouples a rating matrix R into two lower-ranked matrices, a user matrix U and a book matrix B. The rating matrix R is inherently sparse, and our goal is to predict these “empty cells”. During training, we hold one matrix constant and find a square loss minimizer with respect to the other matrix, and then iteratively switch the roles. This matrix computation can be parallelized based on the number of blocks specified.

※ For our project, we set the parameter, "coldStartStrategy" to "drop", so that the unseen users in the validation/test set are dropped at the prediction time.

|| Evaluation Metrics: NDCG, MAP

Our primary focus is on NDCG (Normalized Discounted Cumulative Gain) and MAP (Mean Average Precision), offered by Spark MLlib Ranking Metrics module. We care more about whether a user actually consumes the recommended items or not (0/1 loss), and less about how much the prediction is off. In this sense, Regression Metrics, such as RMSE and MAE, are less appropriate as they focus more on individual rating predictions. Also, the order matters. By displaying items with higher recommendation scores on top, we can expect higher likelihood of a user choosing the items. Thus, MAP and NDCG are more relevant as they take the recommendation order into account while Precision at K does not.

|| Hyper Parameter Tuning: Rank, Regularization

- Rank: the dimension of latent factors
  - ranges: [10, 15, 20]
- Regularization: the regularization parameter to control overfitting
  - ranges: [0.01, 0.05, 0.1, 0.3, 1]

* measured validation score on 1% subsample of filtered data (same as the data for Extension 1)

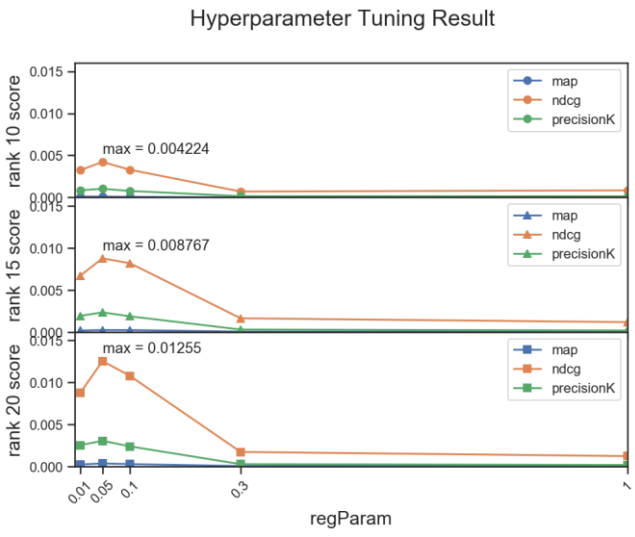

Figure 4. Ranking Metrics for Each Hyperparameter Set

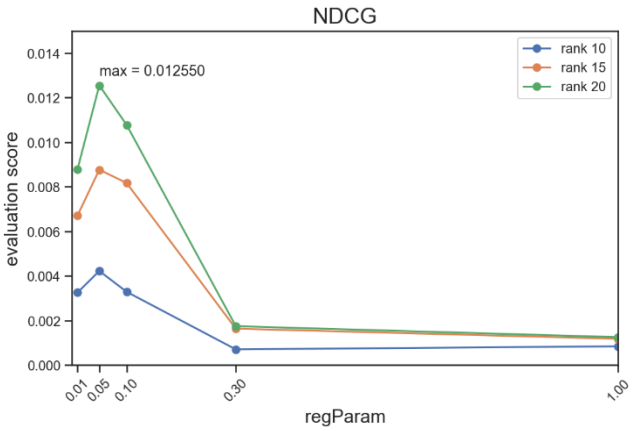

Figure 5. NDCG Comparison

The performance may have been better if the model is trained on larger sample, but due to the cluster overload we performed the grid search on the 1% subsampled validation set over the above mentioned two parameters, rank and regularization. The graph comparing their NDCG, MAP, and Precision at k is shown in ***Figure 4***. The best models based on NDCG, MAP, and Precision at K respectively all had the same hyperparameter setting. The best hyperparameter setting against the validation set was rank = 20, regParam = 0.05, with the resulting NDCG score of 0.01255, MAP of 0.00038, and precision at K of 0.00307.

|| Generalization Result (performance on test set)

Rank	regParam	NDCG	MAP	Precision at 500
20	**0.05**	0.0129	0.0005	0.0032

Figure 6. Performance on Test Set

Our model with the best hyper-parameter set is well-generalized as it can be seen from the similar performance scores on validation and test set. Test result on the larger set (10% subsample) will be discussed in **4. Extensions 2** along with NMSLIB comparison.

4. Extensions

Extension 1. Model with Filtered Data

Two modification on the dataset were discussed in **Section 2. Data Processing**. Here, we summarized the performance results.

|| Accuracy Improvement

Application of filtered data on the model provided higher NDCG and MAP score compared to the baseline model using unfiltered data, even though dropping users with few interactions might have led to information loss, which could have negatively impacted the model performance to some extent. One possible explanation is that dropping unread book data alleviated the data skewness that comes from zero-ratings and thus improved overall accuracy.

Methods	MAP	NDCG	Precision at 500	RMSE
Unfiltered Data (Base Model)	0.00004	0.0016	0.0004	1.9348
Drop Unread & Interactions < 30	**0.00007** (1.75x ↑)	**0.0033** (2x ↑)	**0.0008** (2x ↑)	**1.4025** (0.72x ↓)

Figure 7. Model Performance with Unfiltered/filtered data (default settings; rank=10, regParam=0.1; validation score)

Extension 2: Fast Query using NMSLIB

|| Approximate Nearest Neighbor

The brute-force approach to find inner products between user-item vectors, can be time-consuming, albeit exact, especially when dealing with a large dataset. Here, Approximate Nearest Neighbors(ANN) can speed up queries by preprocessing the data into an efficient index and searching smaller candidate sets instead of all pairs. There are various similarity-indexing libraries available, such as Annoy, FLANN, Facebook's Faiss, Yahoo's NGT, and NMSLIB.

|| NMSLIB (Non-Metric Space Library)

Anirudh K., Meher K., and Siddha G. provide in their book [1] recommendation on which ANN libraries to choose under various scenarios. Our setup belongs to the first scenario, and the use of Annoy or NMSLIB is recommended. Of the two libraries, NMSLIB outperforms Annoy for every recall rate, according to ANN-Benchmarks [2]. Also, we choose to use HNSW indexing as it is the most powerful option among the methods available in NMSLIB.

|| Query Time and Performance Comparison

	NMSLIB	Brute-Force
20% Subsample Test Set	· Time: 13m 56s · NDCG: 0.0013 · MAP: 0.00003	· Failed (OutofMemoryError)
10% Subsample Test Set	· **Time: 5m 50s** · NDCG: 0.0023 · MAP: 0.00006	· Time: 33m 02s · **NDCG: 0.0053** · **MAP: 0.0011**
1% Subsample Test Set	· **Time: 3m 32s** · NDCG: 0.0098 · MAP: 0.0004	· Time: 4m 23s · **NDCG: 0.0129** · **MAP: 0.0005**

Figure 8. With(out) NMSLIB (rank=20, regParam=0.05; test score)

Interestingly, there was less benefit for using NMSLIB for 1% subsamples, while it clearly outperformed the brute-force for larger sample case (10% subsamples) in terms of query time. We found out that, when using NMSLIB, the searching process by itself is fast, but the bottleneck is the indexing process. In other words, with a small dataset, the benefit of indexing (fast query results) does not outweigh its cost.

Moreover, while the brute-force with 20% subsamples failed due to out-of-memory error, our NMSLIB approach succeeded, showcasing that it is also beneficial in terms of memory usage efficiency.

Generally, our results are consistent with the theory that ANN's prediction would always underperform the brute force approach, as they are "approximate measures" scanning only parts of the all possible pairs.

Contributions

[Hyun Jung] Data Processing, Model Building, Evaluation Code, Hyper-parameter Tuning, Code Merge/Finalizing
[Sujeong Cha] Data Processing, Model Building, Evaluation Code, NMSLIB, Report Finalizing

References

[1] Koul, A., Kasam, M., & Ganju, S. (n.d.). Practical Deep Learning for Cloud, Mobile, and Edge: Real-World Ai & Computer-Vision Projects Using Python, Keras & TensorFlow.
[2] GitHub. 2020. Erikbern/Ann-Benchmarks. [online] Available at: <https://github.com/erikbern/ann-benchmarks> [Accessed May 2020]

[Code Reference for Recommendation using NMSLIB] GitHub. 2020. Benfred/implicit. [online] Available at: <https://github.com/benfred/implicit/blob/4dba6dd90c4a470cb25ede34a930c56558ef10b2/implicit/approximate_als.py#L37> [Accessed May 2020]

Appendix

1. Hyper-parameter Tuning Results (Filtered + 1% Subsampled)

Rank 10

regParam	MAP	NDCG	Precision at 500
0.01	0.0001	0.0033	0.0009
0.05	**9.7976e-05**	**0.0042**	**0.0010**
0.1	7.5447e-05	0.0033	0.0008

Rank 15

regParam	MAP	NDCG	Precision at 500
0.01	0.0002	0.0067	0.0019
0.05	**0.0002**	**0.0088**	**0.0023**
0.1	0.0002	0.0082	0.0019

Rank 20

regParam	MAP	NDCG	Precision at 500
0.01	0.0003	0.0088	0.0026
0.05	**0.0004**	**0.0125**	**0.0031**
0.1	0.0003	0.0108	0.0024

2. ANN library recommendations (Reference [1])

Table 4-3. ANN library recommendations

Scenario	Recommendation
I want to experiment quickly in Python without too much setup but I also care about fast speed.	Use Annoy or NMSLIB
I have a large dataset (up to 10 million entries or several thousand dimensions) and care utmost about speed.	Use NGT
I have a ridiculously large dataset (100 million-plus entries) and have a cluster of GPUs, too.	Use Faiss
I want to set a ground-truth baseline with 100% correctness. Then immediately move to a faster library, impress my boss with the orders of magnitude speedup, and get a bonus.	Use brute-force approach

3. ANN Benchmark (Reference [2])