DS-GA 1004 Final Project Report

Jialing Xu(jx1047), Lin Jiang(lj1194)
Team rin733, Center for Data Science, New York University
Github: https://github.com/nyu-big-data/final-project-rin733

1 Basic Recommendation System

1.1 The Implementation Process

To build a basic recommendation system, we finished the following tasks in order: Downsampling training data; Indexing datasets using the same indexers ; Fitting ALS model; Hyperparameter tuning; Evaluating on test set; Developing extensions; Repeating tuning and testing. Details of each step are explained as follows:

1.1.1 Downsampling

To quickly prototype our model, we generated a downsampled training set. Throughout the project, this is the only training set we used. Because we were not able to save the whole indexed training parquet due to out of memory error or randomly killed jobs on Dumbo, and training on the whole dataset takes longer time. We used the full validation and test set though. To make sure our ALS model be evaluated properly, we first selected all records of users that exist in the validation and test set. Then random sampled 5% of the training set without replacement. Finally we combined these two dataframes and dropped duplicates rows. We ended up with a dataset that contains about 7.76% of the original given training data.

1.1.2 Indexing

We built two indexers on user_id and track_id with a pipeline. This pipeline was fitted on downsampled training data first, and then it was used to transform downsampled train set, complete validation and test datasets. We encountered out of memory error when trying to save the transformed train set as parquet file. But this problem was solved after we converted the indexed ids from long type to integer type, and repartitioned on user_id with higher number of partitions.

1.1.3 ALS model fitting

We used implicitPrefs=True and coldStartStrategy="drop" for ALS model. We experimented with repartitions on user_id column and cache intermediate results to make sure our fitting process goes smoothly. The fitting went well at default maxIter = 10. But we found it hard to increase this number without getting errors even when combined with using checkpointInterval under cluster deploy mode. In rare cases when we succeeded at maxiter = 15 and 20, both validation and test results became better than maxiter = 10, under the same parameter settings for rank, regParam, alpha.

1.1.4 Hyperparameter tuning

We used Mean Average Precision for both hyperparameter tuning and test set evaluation. The reason is that for recommendation system in implicit feedback setting, what matters is not the exact number of count for each user but whether we can effectively predict the most likely interaction between a user and

an item. This rules out RMSE since we are evaluating a binary result (i.e. with or without interaction). While AUC shows overall goodness, it does not distinguish between early position and late position in the ranked list. Since we only cares about top K items, we decided to use the most popular metric MAP which assigns a relevance score defined by indicator function of interaction, so takes into account the order of recommended items.

Since cross validation is not feasible typically for recommender system due to problematic unseen user in validation set, we used for loop to do hyperparameter tuning. Apart from manually controlled grids which we used to get an idea of general landscape at different scales, we also tried randomized grids since they reveal the rough range of performance more efficiently. The range that we searched for each parameter was: $\alpha \in [0.1, 35]$, regParam $\in [0.01, 4]$, rank $\in [2, 200]$ for manual tuning; $\alpha \in [0, 35]$, regParam $\in [0, 10]$, rank $\in [10, 100]$ for randomized search. Several discoveries we made are listed below.

1.2 The Evaluation Results

It's worth noticing that we got most results below by running maximum iteration of 10 (default value) which we suspect is not enough for the model to converge. We found that the validation performance continues to increase as rank exceeds 200, which might be due to this max iteration restriction that is hard to break.

In general, we found that larger rank is always better when the other two are fixed to be small. But increasing α allows low rank model such as rank50 to have descent results like MAP around 0.043. α has an impact on performance, the ideal range is around 29 which indicates that enhancing count signal is helpful. The performance with respect to changing regParam is not monotonic, we have multiple local maximizers of regParam at several range.

Our final best setting is when rank = 90, regParam = 4 and $\alpha = 30$, which got validation and test MAP around 0.049 when maxiter = 10 and test MAP of 0.052 when maxiter = 15. We have put our complete hyperparameter tuning results under the outputs folder in the github repository.

2 Extensions

2.1 Alternative Model Formulation

The original distribution of count is really skewed: it has mode at count = 1 and a huge range from 1 to 9667. We may not be confident about low count interactions, so we set all count <=1 to zero (dropone) and set all count<=2 to zero(droptwo). Additionally, we found that log2 transformation gave us a moderately balanced distribution, so we also tried log2 transformation on count then converted the resulting values into integer type to further bucket the counts (log2_int). And we also tried log2 transformation without type conversion(log2). With these four strategies to normalize the count signal and reduce complexity, we repeated the hyperparameter tuning process mentioned above.

By comparing locally, we found that relatively bigger α is still better for the performance in all four transformed datasets which means model is in need of stronger count signals. But globally, the performance of these transformed datasets after grid search were not as good as the original dataset.

Higher rank couldn't buy them more precision as it was in original dataset. And the best result we can get from alternative models is less than 0.04.

2.2 Fast Search

We use NMSLIB to implement accelerated search at query time. According to this benchmark, nmslib using Hierarchical Navigable Small World Graph (HNSW) algorithm dominates other methods in low recall region and remains to be competitive at high recall region. This motivates us to integrate this tool with our recommendation system of large number of users and items to accelerate finding the most similar item for each user in terms of maximum inner product.

To install NMSLIB with Python bindings on Linux, OSX and Windows, it suffices to run **pip install nmslib.**

Note that NMSLIB does not have out of box inner product as similarity metric. We have to use cosine similarity as surrogate and do some order preserving transformations on the original data to make these two search exactly equivalent. One approach is proposed in this paper.

We selected a moderately good model (rank = 80, alpha = 20, regParam =0.1), exported this model to our local machine. For any ALS Model, the userFactors (U) and itemFactors (V) contain the low rank matrices that we use to compute inner product, where each row of U or V represents the factors of a particular user or item respectively. The transformation is the following: for each user factor u_i, the transformed factor becomes $\tilde{u}_i = (u_i{}^T, 0)^T$. For each item factor v_i, suppose $\phi = max_i \|v_i\|$, then the transformed factor becomes $\tilde{v}_i = (v_i{}^T, \sqrt{\phi^2 - \|v_i\|^2})^T$.

This way, we have $j = argmax_i \tilde{u} \cdot \tilde{v}_i = argmax_i \frac{u \cdot v_i}{\|u\| \|v_i\|}$.

Table 1: NMSLIB vs BruteForce

	Query Time /s	**Seconds per query**	**MAP**	**Index Building Time (nmslib) /s**	**M**	**efConstruction**	**efSearch**
BruteForce	2847.8	0.02848	0.0413	n/a	n/a	n/a	n/a
NMSLIB	49.0	0.00049	0.0382	63.1	20	300	800
	61.7	0.00062	0.0383	68.8	20	300	1000
	47.3	0.00047	0.0386	927.1	50	3000	800
	49.7	0.00049	0.0386	171.4	20	500	1000
	40.2	0.00040	0.0386	1397.8	100	500	800

Our query task is to recommend the top 500 items for each user in the test set. We implemented a brute force method in numpy array to calculate the inner product of each user factor with all item factors and sort out the top 500 items. To use nmslib we first build an indexer using item factors data. Then we used batch query to get recommendations for all users. We ran brute force once and accelerated search several times, adjusting the index and query parameters to see the tradeoff between precision and speed. We recorded the total query time and MAP for both methods as well as the time taken to build index for nmslib. MAP was computed by converting the recommended lists back to pyspark dataframe to use pyspark Ranking Metrics. Some results are listed in Table 1.

As can be seen from the table, query time of approximate similarity search is about 57 times faster than brute force exact search. MAP by approximation is slightly lower but not too bad. And there is a slight tradeoff between MAP and query time for nmslib. Also with longer index building time, query time will be faster while MAP not getting worse, although the speed gained might not be worth the wait time to build index.

3 Reference

[1] Non-Metric Space Library (NMSLIB), GitHub repository, https://github.com/nmslib/nmslib
[2] Benchmarks of approximate nearest neighbor libraries in Python, GitHub repository, https://github.com/erikbern/ann-benchmarks
[3] Malkov, Yury A., and Dmitry A. Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs." *IEEE transactions on pattern analysis and machine intelligence* (2018).
[4] Bachrach, Yoram, et al. "Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces." *Proceedings of the 8th ACM Conference on Recommender systems*. ACM, 2014.

4 Contribution

We collaborated on every part of this project:

- Jialing Xu (jx1047): basic recommender system, extensions
- Lin Jiang (lj1194): basic recommender system, extensions

5 Codes List

Please find all below codes in our github repository:

- Downsample_train_set.py: downsampling on full train set
- Fit_indexer_int.py: indexing datasets using the same indexers fitted on train
- Recsys_train_*.py: model training code
- Recsys_test.py: model testing code
- Extension_1_modify_count.py: applying thresholding/logarithm transformation to raw data
- Extension_2_lsh_nmslib.py: implementing LSH