DSGA-1004 Final Project: MovieLens Recommender System
New York University, Center for Data Science

Guilherme Albertini, Giacomo Bugli, Luigi Noto*

Abstract

We build and evaluate a collaborative-filtering based recommender system with Spark alternating least squares (ALS) implementation[1] using the MovieLens dataset[2]. We compare the Spark's parallel ALS model to `lenskit` single-machine implementation in terms of efficiency and model performance. Finally, we implement accelerated search at query time using `annoy` spatial data structure, and compare this fast search method to the brute force approach in terms of query efficiency and quality of recommendation.

1 Introduction

Our group aimed to develop a collaborative filter recommender system based on explicit feedback using the MovieLens datasets (small and full version). We partitioned the data into training, validation and test sets. The model has been fitted on the training set using PySpark's alternating least squares (ALS) implementation, tuned on the validation set and finally evaluated on the test set. Before developing the collaborative filter we implemented a conventional popularity baseline model as a standard practice. The popularity baseline has also been fitted, tuned and evaluated on the same datasets to obtain a baseline score for our collaborative filter.

2 Project Structure Overview

To give the project more of a polish, the authors consulted previous work samples and Open Source projects to format the online portion of the assignment using packages that would later be sent to the cluster for worker nodes to access. The shell setup script was slightly modified to include the zipped package files that will be discussed below.

2.1 Wheel, Dist and Build

To make the code more modular (instead of having a stuffy main routine or mishmash of independent scripts) the design choice of packaging modules for the different user-defined classes and data processing methods better enforced an object-oriented approach. No requirements.txt document was included as the limitations of installing packages that may be unavailable on certain clusters or were restricted from being installed through pip made it a burden. Linting and other polishes were

*Chief responsibilities of: **Guilherme** - ALS validation, Baseline Code Structure, Report Writing; **Luigi** - Data Splitting Algorithm, ALS validation, Extension 2, Report Writing; **Giacomo** - Data Splitting Algorithm, Popularity Baseline, Extension 1, Report Writing.

[1] `https://spark.apache.org/docs/3.0.1/ml-collaborative-filtering.html`

[2] `https://grouplens.org/datasets/movielens/latest/`

not implemented due to a non-unanimous vote of confidence regarding their merit. More details can be found in `README.md`.

2.2 Dataset Splitting

As for the data partitioning, we thought that the model needed some history of interactions for each user when being fitted, to avoid the case of getting random embeddings for some users if they are not represented during training. To make sure this happened, we decided to implement a time-based partitioning of each user's history, where the 80% least recent interactions for that user fall in the training set and the 20% most recent fall in the validation/test set, then interpreting the observations in the validation/test set as "future" interactions of the users in the dataset. In order to construct the validation and test set, we considered the fact that the validation set is used for hyperparameter tuning and the test set is used only at the end to get an estimate of the model's generalization performance. Thus, to get a better estimate of generalization performance, we split the data consisting in the 20% most recent observations of each user in such a way that no user had observations in both the validation and test set.

2.3 Models

The popularity baseline model and ALS model were called in the main routine using methods found in their respective packages. Namely, those in the validated models folder contained the class scripts functioning to select the best model using the validation data and assess performance on the test set. This way, the user could easily call which parameters could be tested and which dataset sizes should be considered for a run in the cluster.
The baseline popularity model created a class implementing a standard popularity model, which gets the utility matrix containing users' ratings and computes the top most popular movies, where popularity is defined as the average rating for each movie. For model regularization two hyperparameters can be tuned, the threshold which determines the minimum number of ratings a movie should have in order to be considered for the popularity computation, and the damping factor which is equivalent to adding extra observations with 0 rating. To do so, the class contains a fit method and an evaluation method. Further details are found in `README.md`.
The alternating least squares model (ALS) created a class wrapper around the one provided by Pyspark in order to make a streamlined process of hyperparamter tuning and a more robust way to evaluate metrics in fewer lines of code in main.py, thereby including evaluate and validate methods in the `ValidatedALS` class. The user must simply pass in arrays of hyperparameters for cross validation, the split datasets generated previously to the `ValidatedALS` class and select the evaluation metric to be considered. The class computes the metrics in the evaluate method using a RankingEvaluator class in the `evaluate` method, while in the `validate` method performs validation for the sets of parameters provided and returns the model fitted on the best parameter configuration.

3 Results

3.1 Choosing Evaluation Criteria

When producing a list of recommendations we should aim at reducing the error that we make in the first few elements rather than treating all the errors with the same weight, since most of the the users consider only the first couple of recommendations. To achieve this we need a metric that

weights the importance of the errors accordingly. We then want a metric that highly penalizes the errors at the top of the list and gradually decrease the importance of the errors as we go down the list. With this goal in mind we then considered two metrics to use for model evaluation and comparison: Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG).
The MAP does this by computing the Average Precision (AP) for each user and then averaging them. The issue with this metric is that it is unable to handle fine-grained numerical ratings, and thus when performing evaluation it is necessary to threshold the ratings to make them binary. The NDCG achieves the same end goal of the MAP but it takes into account the granularity of the ratings, going beyond the binary case scenario.
Taking this into account we choose the MAP as main metric to perform model evaluation. The reason behind it is that we are considering the movies in the test set with rating higher than 2.5 as ground truth (hence movies that the user has actually enjoyed) and compute the above metrics comparing the set of recommendations and such ground truth. Therefore, we are applying a binary criterion where a movie recommendation is either right (included in the ground truth) or wrong (we consider movies that the user watched but didn't enjoy as a wrong recommendation), also taking into account the order of the recommended movies.

3.2 Evaluation of popularity baseline on small and full datasets

We evaluated the popularity baseline on both the small and full datasets, finding that the baseline model on the small dataset had an optimal configuration consisting of damping factor of 20 and the full dataset had an optimal damping factor of 40. The validation has been performed for the following set of damping factors: [0, 5, 10, 15, 20, 25] for the small set and [10, 20, 30, 40, 50] for the full dataset version. The best results are shown in Table 1.
From Table 1 we can see that the optimized model had better scores on the NDCG@100 metric over MAP@100 for both the small and the full data sets. Moreover, we notice that the small dataset has better outcomes in terms of MAP than the full dataset. The reason behind it may lie on the metrics used (we highly value correct recommendations early in the list) and the fact that on the small data set the number of users is much smaller than on the whole set. Therefore, even if there's a greater chance to randomly recommend a correct movie on the full data set, we have that the higher number of users most likely implies a broader difference in tastes, and with less users the prediction is more likely to be overfitting (we train on all users).

3.3 Latent factor model's hyper-parameters

Our latent factor model used custom code alongside Pyspark's Alternating Least Squares (ALS) model. The parameters we used for model selection were the rank, regularization penalty, and max iterations allowed until convergence. Other parameter values were left with their default settings.
The rank parameter controls the rank of the factorization; these are the presumed latent or hidden factors. The regularization parameters vary the penalty scaling the L2 norm sum of the user-factors vector and item-factors vector for every user-item pair. The max iterations we vary through can be thought of as the epochs for training. We show resuts of the hyperparamter tuning for ALS in Table 2.

3.4 Evaluation of latent factor model on small and full datasets

We evaluated the latent factor model on both the small and full datasets, finding the best configurations as displayed in Table 2. The validation has been performed for the following set of

Table 1: Cross-Validated Best Popularity Baseline Results (No Threshold)

Data Size	Damping Factor	MAP@100	NDCG@100
Small (Train)	20	0.1023	0.3228
Small (Test)	20	0.0524	0.1073
Full (Train)	40	0.0431	0.1566
Full (Test)	40	0.0493	0.1794

Table 2: Cross-Validated Best ALS Results

Data Size	RegParam	Rank	Max Iter	MAP@100	NCDG@100
Small (Train)	0.01	30	20	0.1232	0.3541
Small (Test)	0.01	30	20	0.0661	0.1621
Full (Train)	0.1	50	35	0.1795	0.2659
Full (Test)	0.1	50	35	0.0832	0.1973

parameters for the small dataset: `rank` $= [10, 20, 30, 40]$, `regParam` $= [0.001, 0.01, 0.1, 1]$, and `maxIter` $= [10, 15, 20, 25, 30]$; while for the full dataset: `rank` $= [20, 30, 40, 50, 60]$, `regParam` $= [0.001, 0.01, 0.1, 1]$, `maxIter` $= [25, 30, 35, 40, 45]$. The best results are also shown in Table 1.
From Table 2 we can see that the optimized model had better scores on the NDCG@100 metric over MAP@100 for both the small and the big data sets. Moreover, we can notice that the big dataset has better performances in terms of MAP than the small one. This is consistent with expectation as with the larger dataset the model can train the users and item factors on more observations.

4 Extensions

4.1 Extension 1: Single-Machine Implementation

In this extension, we compared the performances of the PySpark's ALS parallel implementation obtained before with the performances of the ALS model on single machine on the same exact data splitting (train-validation-test). First, to implement the ALS on local we used the library `lenskit`[3]. This library was chosen because it yields a set of tools easily implemented in Python to mimic parallelized computations on your local machine, and provides great flexibility thanks to its many interfaces. The code implemented substantially aims at mirroring what we have done on PySpark, by implementing a wrapper function for the `lenskit` methods to perform model validation and a main routine that calls such function. For the sake of comparison, both implementations (remote and local) have been validated on the same set of hyperparameters to ensure fair treatment, and evaluated with the same metrics. The comparison as been conducted both in terms of model fitting time as function of data size (small and full) and in terms of accuracy using Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG). To time the runs, we compute the average fitting time over the set of hyperparameters in the validation. This is done for both implementations in order to obtain more comparable and robust results of models' efficiency. For each implementation/data size combinations we show the resulting measures of efficiency and their respective performance in Table 3.
As we can see the `lenskit` implementation performs worse than the parallelized version in terms

[3]More information on how it works: `https://lkpy.readthedocs.io/en/stable/index.html`

Table 3: Single Machine Comparison: Lenskit ALS vs PySpark ALS Results

Data Size	Model	MAP@100	NCDG@100	Time to Fit (s)	Reg Param	Rank	Max Iter
Small (Test)	Lenskit ALS	0.04204	0.06529	0.6912	1	40	10
Full (Test)	Lenskit ALS	0.05498	0.04355	245.7438	1	50	30
Small (Test)	PySpark ALS	0.0661	0.1621	16.0586	0.01	30	20
Full (Test)	PySpark ALS	0.0832	0.1973	398.6661	0.1	50	35

of accuracy for both data sizes, while in terms of efficiency the single machine implementation performs better than the Spark one on both data sizes. Moreover, as we expected, model fitting on the small set requires less time on the small data set than on the full one. As regards efficiency (time to train the model) one of the reasons could lie on the fact that `lenskit` is specifically optimized for parallelizing jobs[4] on local machine for which the data fit in memory, and that the modularity of the approach allows to have faster computation. In general we can have that with very large data sets that exceed single machine's memory capabilities, Spark offers faster run time and greater scalability from multi-core parallelism and better execution engine. In terms of accuracy, both implementations seem to have very similar performances (with PySpark slightly better than lenskit) differing of a centesimal unit. Further analysis would be required to asses the two approaches performances as different hyperparameters and larger data sizes are used.

4.2 Extension 2: Fast Search

In this extension, using the user and item factor matrices obtained by fitting the ALS model with the best hyperparameter configurations on the full dataset, we implemented accelerated search at query time with the help of a spatial data structure. We then compared the fast search implementation to the brute-force method, consisting in computing the inner products of the query (user latent factors) with all the item factor representations and getting the items with the k highest inner products, where k is the number of recommendations to provide. The `annoy`[5] library, used by Spotify for music recommendations, has been used to implement the fast search method. It is a tree-based method for approximate nearest neighbor search, which consists in creating an index by building a forest of trees, where each tree is constructed in the following way: we pick two points at random and then split the space by the hyperplane equidistant from those two points; we then keep splitting each subspace recursively until the number of points associated with a node in the tree is small enough. At query time, the forest is traversed to obtain a set of candidate points, among which the the k closest to the query point are returned, where k is the number of recommendations[6]. There are two parameters to tune in Annoy that control the accuracy-efficiency trade-off:

- `n_trees`: this parameter is provided at index build time and affects the building time and the index size; it consists in the number of trees we build.
- `search_k`: this parameter is provided at query time and affects the search performance; it consists in the number of total nodes that will be searched; a larger value will give more accurate results, but will take longer time to return the return the query; if you set this parameter to a very large value, you may essentially end up with exhaustive search.

[4] See `https://lkpy.readthedocs.io/en/stable/performance.html`

[5] `https://github.com/spotify/annoy`

[6] More information on how Annoy works: `https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html`

Table 4: Query Methods Comparison: Brute Force vs Annoy

	`n_trees`	`search_k`	Index building time	Queries per second	Avg Recall
Brute-Force	–	–	–	∼13	1.0
Annoy	100	100	∼2s	∼18043	0.002
Annoy	100	1000	∼2s	∼6598	0.016
Annoy	100	10000	∼2s	∼841	0.196
Annoy	100	50000	∼2s	∼232	0.635
Annoy	100	100000	∼2s	∼116	0.865
Annoy	500	100	∼11s	∼4504	0.002
Annoy	500	1000	∼11s	∼2737	0.023
Annoy	500	10000	∼11s	∼588	0.184
Annoy	500	50000	∼11s	∼189	0.653
Annoy	500	100000	∼11s	∼105	0.879

In order to highlight the efficiency gains of the fast search method, we produced 100 recommendations for the first 100 users[7] in the dataset using Annoy (with respect to the inner product distance) with different values of `n_trees` and `search_k` in order to find the right balance in terms of average recall (the fraction of true nearest neighbors found, averaged over all queries) and number of queries per second with respect to the brute-force method. Some of the results[8] are shown in Table 4. In particular, three promising configurations are the following: (1) `search_k` = 10000 with about 841 queries per sec and 0.196 average recall, (2) `n_trees` = 100, `search_k` = 50000 with about 232 queries per sec and 0.635 average recall, (3) `n_trees` = 100, `search_k` = 100000 with about 116 queries per sec and 0.865 average recall. The first prioritizes efficiency to correctness of recommendations, being able to generate about 65 times more queries than brute force per sec with about 20% of correct recommendations (20 out of 100 movies in our case). The third prioritizes correctness of recommendations, at the expense of having a lower increase in queries per sec with respect to brute force. The second configuration is between the other two with a more balanced trade-off. These three configurations show that by using a clever data structure we can increase query time efficiency by much (depending on our needs) with respect to exhaustive search while still being able to provide a subset of the "true" meaningful recommendations according to the estimated user and item embeddings.

5 Concluding Remarks and Future Work

Further hyperparameter tuning would be beneficial using the full dataset in regards to Popularity Baseline and the ALS models. Due to time and Spark cluster configuration knowledge constraints, the authors see the most refinements would come from this area. Attempting a random search for hyperparamter tuning over a grid-search-esque algorithm may also prove more efficient, though further investigations have to be done to assess its outcomes over the provided nested loop optimization. Regarding the fast search extension, a more comprehensive experiment on all the users instead of just the first 100 and generating a higher number of recommendations would be beneficial for the comparison to further investigate the efficiency gains. It would also be interesting to

[7]We did not generate recommendations for all the users in the dataset for this experiment due to resource constraints, since the comparison has been run on a local machine.

[8]For all the results, see `fast_search_results.txt` in the github repository.

compare the Annoy fast search method to other methods such as NMSLIB or ScaNN and verify which method turns out to be the most efficient in our application.
It also became quite clear that the authors had limited knowledge of how to properly configure the Spark cluster (i.e. interactions between Spark, YARN scheduler, JVM, etc.) and were in need of a lot of technical assistance. Should a more rigorous treatment of the Spark cluster environment be required prior, we predict that only a fraction of the memory-bound errors experienced would ever come to be. Though the competition for the authors' time during these trying last weeks proved challenging, greater efforts spent on how to best configure the cluster environment (e.g. by allocating adequate memory to the driver, or computing the number of and memory for each executor, or tuning the YARM AM container etc), one would expect to see improved job run times and better maximize cluster throughput. Unfortunately, this detracted from actually building the basic recommender system.