Movie Recommendation Systems Built with MovieLens Dataset

Zhehan Shi [1] **Zixiang Pei** [2]

1. Introduction

Recommendation systems has been all around us. When we watch movies, listen to music, or order takeouts, we are all exposing our personal information, which allows companies to analyze our preferences and recommend items alike for better promotion and user engagement. In this project we aim at building a recommendation system with the MovieLens dataset using various models, including popularity baseline model and latent factor model using Spark's alternating least squares (ALS) method, and try to find the best hyper-parameters for best recommendation.

2. Overview

2.1. The MovieLens Data

The dataset describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service. The small dataset contains 100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users. The large dataset contains 27,000,000 ratings and 1,100,000 tag applications applied to 58,000 movies by 280,000 users. (2)

2.2. Train-Val-Test Split

When running traditional machine learning algorithms, we would keep disjoint sets of training-validation-test sets. We train on the training set, tune hyper-parameters with the validation set and test the model's final performance with the test set. For building recommendation systems, however, we need to have at least some movies in the training set for users in both the validation set and the test set. Furthermore, since we don't want the unlucky scenario in which users in the validation or test set have none of their movies sampled to the training set, for those users and movies that have less than 20% of the mean ratings count, we automatically put them into the training set. After we split data in this way the data was roughly 60% for training, 20% for testing and 20% for validation. We also pre-processed out the timestamp feature and decided to not use it in our model. To illustrate the idea in a more straight forward way, we attach a demo below:

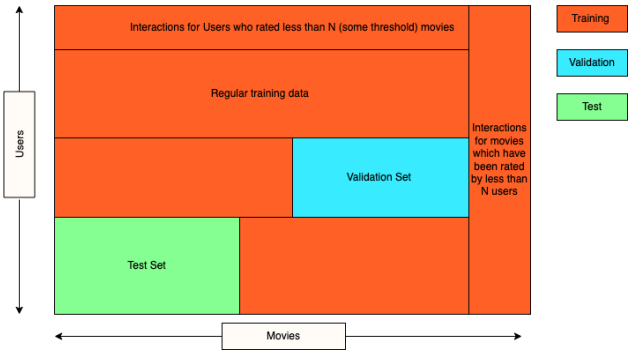

Figure 1. Graph Demo of Data Split Procedure

3. The Recommendation Models

3.1. Popularity Baseline Model

After we split the data we first ran our baseline model. The baseline model, according to what the professor has introduced, is just the average ratings of each movie for those people who have actually watched the movie and the recommendations are just the 100 highest-rated movies and these recommendations are the same for everyone.

In this process, we notice that some movies have only very few people rating them and if we take into account those ratings, the recommendations could very well be biased. With regard to this, we decide to only take into account the movies that have more than say M (a threshold) ratings. We use the validation set to tune this hyper-parameter M and use the testing set to evaluate our result.

For evaluation criterion, we have looked at several different aspects. The precision-based metrics focus on how many of the recommended documents are in the set of true relevant documents. The difference between precision and MAP is that MAP takes the order of the recommendations into account. The NDCG (Normalized Discounted Cumulative Gain) metric is similar to MAP and also takes into account the order of the recommendations.

[1](zs1113@nyu.edu, Net ID - zs1113) [2](zp2123@nyu.edu, Net ID - zp2123). Correspondence to: Zixiang Pei <zp2123@nyu.edu>.

Baseline Result	Train	Validation	Test
Small	0.0323	0.0917	0.0919
Large	0.0549	0.0485	0.0484

Table 1. Mean Average Precision for Baseline Model

Baseline Result	Train	Validation	Test
Small	0.1362	0.2722	0.2742
Large	0.1842	0.1900	0.1896

Table 2. NDCG for Baseline Model

Baseline Result	Train	Validation	Test
Small	0.0596	0.1638	0.1498
Large	0.0813	0.0917	0.0913

Table 3. Precision for Baseline Model

3.2. Latent Factor Model

Latent factor model is one of the most commonly used techniques in model-based collaborative filtering. In this project we used Spark's alternating least squares (ALS) method to learn latent factor representations for users and items. For evaluation criterion, we have looked at several different aspects. In addition to the three criteria we used for the baseline model, we also use the RMSE (rooted mean squared error) metric which looks at the standard deviation of the residuals between each movie's true ratings and the predictions.

3.3. Hyper-parameter Tuning

For the alternating least squares method the two most important hyper-paramters are Rank, which is the number of latent factors in the model, and regParam, which specifies the regularization parameter in ALS. In the limited time, we also did a few experiments with the maxIter parameter to ensure better convergence. We start with a large range on the small dataset and gradually narrow our targets down to a few check points and used grid search on the full dataset. Below is a table of "grids" we used:

Table 4. Grid Search Hyper-parameter Search Space

HYPER-PARAMETER	DEFAULT	SEARCH SPACE
RANK	10	0.001, 0.01, 0.05, 0.1
REGPARAM	1.0	20, 50, 100, 200, 300

The results are recorded in tables 5-8 for different criterion:

reg/rank	20	50	100	200	300
0.001	4e-7	6e-6	3e-5	6e-4	0.0018
0.01	2e-7	7e-6	1.9e-4	0.001	0.0017
0.05	5e-6	9e-5	3e-4	5e-4	7e-3
0.10	1e-4	2e-4	5e-5	1e-3	2e-3

Table 5. MAP @100 Grid Search Results

reg/rank	20	50	100	200	300
0.001	0.938	1.077	1.219	1.251	1.181
0.01	0.853	0.884	0.895	0.890	0.876
0.05	0.822	0.821	0.824	0.830	0.832
0.10	0.819	0.821	0.831	0.830	0.831

Table 6. RMSE @100 Grid Search Results

reg/rank	20	50	100	200	300
0.001	8e-6	7e-5	6e-5	6e-3	0.0177
0.01	4e-6	1e-4	2.7e-3	0.0107	0.01749
0.05	1e-4	0.001	0.0033	0.005	0.00697
0.10	0.0021	0.0037	9e-4	0.003	0.0047

Table 7. NDCG @100 Grid Search Results

reg/rank	20	50	100	200	300
0.001	5e-6	4e-5	4e-5	0.0039	0.0097
0.01	3e-6	6e-5	1.7e-3	6e-3	9e-3
0.05	5e-5	5e-4	0.0013	2e-3	3e-3
0.10	8e-4	2e-3	3e-4	1e-3	2e-3

Table 8. Precision @100 Grid Search Results

We can see from the tables that generally the higher the rank, the better the performances. To avoid overfitting, and to save computation power, we decided to stop at rank = 300. In terms of regParam, we can see that regParam = 0.01 and regParam = 0.001 yield similar results, and 0.01 has smaller RMSE. It indicates that a regParam of 0.01 is more generalizable to unseen data and we chose that as our best hyper-parameter. We then used the 300/0.01 combination to run between 5 to 10 maxIterations, and concluded that running 10 iterations the model converges better and it yielded a RMSE of **0.863** and a Mean Average Precision @ 100 of **0.00428** for the full set. After we obtained this best set of hyper-parameters, we applied it back to the small dataset and we achieved a RMSE of **1.352** and a Mean Average Precison @100 of **0.0283**.

3.4. Analysis of Results

When we first obtained our baseline model, we were quite confident that we can beat it with Spark's ALS method, but

at the end, we were wrong. We did apply a few optimization on the baseline model, aside from removing movies with too few ratings, at inference time, we also try to recommend movies that have over M (a threshold) ratings, so the 100 movies we recommend both have many ratings and have a high score. Aside from our overly good baseline model, the hyper-parameter searching we did on Spark's ALS method was pretty effective. Our model with the selected hyper-parameters gives way better mean average precision and smaller rooted mean squared error than the default setting.

4. Extension: Comparison to Single-machine Implementations

For this extension we want to compare Spark's ALS method to single machine implementations such as lightfm and lenskit. We want to compare both time take while training and precision for the first 100 recommendations. We want to design the subset of datasets such that we can see both overhead(which is relatively not affected as much by data size, and more of a intrinsic complexity of each method) and scalability (which is intimately related to data size) of different methods, so we decided to take [0.01, 0.05, 0.1, 0.2, 0.4, 0.6] portion of data and run the three methods on all of the data, so we obtain 18 pairs of results in total.

4.1. Compare to lightfm

The lightfm implementation has a number of popular recommendation algorithms for both implicit and explicit feedback. It supports various types of loss functions when facing different forms of data. For instance, the logistic loss is useful when both positive and negative interactions are present. For our specific task we chose "WARP" (Weighted Approximate-Rank Pairwise) loss because it maximises the rank of positive examples by repeatedly sampling negative examples until rank violating one is found, and is useful when only positive interactions are present and optimising the top of the recommendation list is desired. By choosing the "WARP" loss we can theoretically obtain a better precision. (3)

4.2. Compare to lenskit

Lenskit is a set of Python tools for experimenting with and studying recommender systems. It provides support for training, running, and evaluating recommender algorithms in a flexible fashion suitable for research and education. The package makes it easy to do cross validation so we implemented a 5-fold cross validation for better performance. The precision would be the average precision from the 5 iterations and the time take would be the average training time. (1)

For the hyper-parameters in all three methods we used the best set from our ALS model, namely [rank = 300, regParam = 0.01, maxIter = 10]. The following plots (Figures 2 & 3) show both time and precision against the sizes of the dataset.

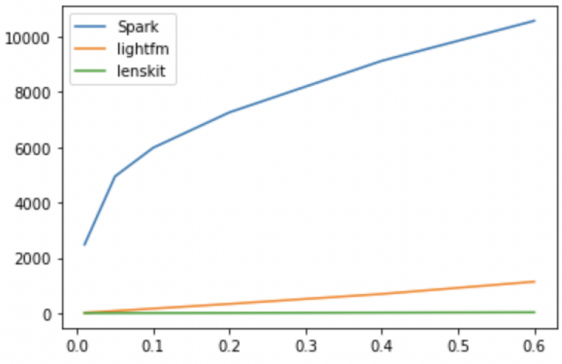

Figure 2. Training Time against Data Size for the Three Methods

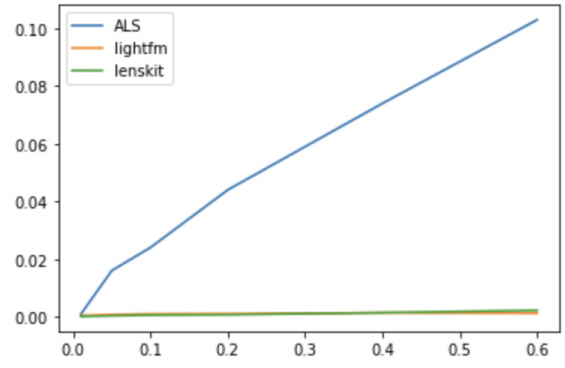

Figure 3. Precision @ 100 against Data Size for the Three Methods

4.3. Conclusion

From the plots we can see that Spark, surprisingly, has much longer training time than both single-machine methods, this could be due to the implementation of our model is not optimized enough and has high overhead. Lightfm took longer than lenskit. It is worth noting that although lightfm and lenskit has shorter training time, they actually has much higher inference time than the ALS model, but since this is not the main concern for this extension we did not include graphs. Another result we can see from the graph is that while training time for lightfm and lenskit increases linearly with the size of data, training time for Spark's ALS method increases slower than linear rate. This shows that Spark is optimizing the algorithm to increase scalability of the method.

In terms of precision we can see that ALS model reaches

0.1 precision @100 with 60% of the data while lightfm and lenskit could not even exceed 0.01 in all 6 datasets.

To conclude we can see from the extension that it is quite surprising that single-machine implementations are actually way faster than Spark's ALS method. This could be due to the large overhead that we haven't optimized enough, and could also be due to the different methods have different training phases and our record is not accurate enough to track down all the training details. In terms of precision, ALS beats both lightfm and lenskit by a large portion, especially as data goes larger and larger. This shows that for large-scale tasks, Spark's ALS method is more powerful with parallel computing and both improves accuracy and has sub-linear increase in time with increasing data size.

5. Discussion

5.1. Conclusion and Future Work

In this project we have done several tasks. We have constructed and optimized a baseline model that only takes into account the mean rating of each movie. We have also constructed a latent factor model using Spark's ALS method and used it to find the best pair of hyper-parameters. Unfortunately, and quite unexpectedly, even the best parameters did not beat our optimized popularity model, and we have a deeper understanding of why it is often best to go directly to the popularity model as a starting point when constructing a recommendation system. Finally, we have also compared Spark's ALS implementation with single-machine implementations like lightfm and lenskit. We found that while the time take at training is longer for ALS model than lightfm and lenskit implementations, the ALS model is more scalable and has much better performance.

If more time is allowed and given more computation power, we will optimize our Spark model further by giving it more parallelism and partition our data in a more efficient way. We will also try out more combinations of parameters and especially run more iterations to ensure better convergence.

5.2. Member Contribution

Zhehan Shi: Main coder for the baseline model and Spark's ALS model.

Zixiang Pei: Main coder for the extension and main writer for the final report.

Design of data split mechanism and hyper-parameter tuning was done together by the two members.

5.3. Code Release

https://github.com/cyberzzhhss/big_data_MovieLens

References

[1] M. D. Ekstrand, "The LKPY package for recommender systems experiments: Next-generation tools and lessons learned from the lenskit project," *CoRR*, vol. abs/1809.03125, 2018. [Online]. Available: http://arxiv.org/abs/1809.03125

[2] F. M. Harper, "The movielens datasets: History and context. acm transactions on interactive intelligent systems (tiis) 5 4 (2015) 1–19. f maxwell harper and joseph a konstan. 2015. the movielens datasets: History and context. acm transactions on interactive intelligent systems (tiis) 5 4 (2015) 1–19," 2015.

[3] M. Kula, "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.*, ser. CEUR Workshop Proceedings, T. Bogers and M. Koolen, Eds., vol. 1448. CEUR-WS.org, 2015, pp. 14–21. [Online]. Available: http://ceur-ws.org/Vol-1448/paper4.pdf