Collaborative-filter Based Recommender System on MovieLens

DS-GA 1004 Big Data: Group 101 - Final Project Report

https://github.com/nyu-big-data/final-project-group_101

Jianyu Zhao
Center for Data Science
New York University
New York, NY
jz5246@nyu.edu

Vanessa Xu
Center for Data Science
New York University
New York, NY
zx657@nyu.edu

Catherine Zheng
Center for Data Science
New York University
New York, NY
rz2432@nyu.edu

1 Abstract

In this Final Project, we built and evaluated a collaborative-filter based recommender system using the MovieLens (Harper and Konstan, "The Movielens Datasets") dataset. Two versions of the dataset are provided: a small sample (*ml-latest-small*, 9000 movies and 600 users) and a larger sample (*ml-latest*, 58000 movies and 280000 users), in which each version of the data contains rating and tag interactions.

We preprocessed the data by partitioning it into training, validation, and testing datasets. Then we applied two statistics models, Popularity Baseline Model and Spark's Alternating Least Squares Model (ALS), to predict the top 100 movie recommendations for users in testing and validation dataset.

As for evaluation, we used two different evaluation metrics, Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG), from *pyspark.ml.evaluation.RankingEvaluator( )* to evaluate the model performance((RankingEvaluator — PySpark 3.1.1 Documentation)). To achieve higher performance, we tuned hyperparameters including rank, regParam, alpha and maxIter in Spark's Alternating Least Squares Model (ALS — PySpark 3.2.1 Documentation):

We then compared our ALS model with single machine implementation—LightFM in terms of both efficiency and accuracy. Lastly, we compared the query search time for querying using brute force algorithm and fast search algorithm. The latter employs the spatial data structure, and significantly improves query search time efficiency compares to the brute force algorithm.

2 Data Pre-processing

In order to evaluate the performance of the recommendation system, we partitioned the data *ratings.csv* into training, validation, and testing sets. We started by splitting the data's user identities with ratio 0.2:0.4:0.4 into training, validation, and testing sets. Then we partitioned the user's interactions in validation and testing datasets based on each user's median timestamp. The older half of each user's interactions in the validation and testing sets goes into the training set, and the newer half remains in validation and testing sets respectively. Then the ratio of the partitioned data for training, validation, and testing becomes 0.6:0.2:0.2.

Table 1 illustrates an example of an interactions table for 5 users. the values are timestamps that the user gives ratings on each movie instead of user ratings from 1: the oldest to 4: the newest. Table 2, 3, and 4 show an example of how the data in table 1 is partitioned into training, testing, and validation sets using the method described above.

	user_1	**user_2**	**user_3**	**user_4**	**user_5**
movie_1	1	1	1	1	1
movie_1	2	2	2	2	2
movie_1	3	3	3	3	3
movie_1	4	4	4	4	4

Table 1: Example of ratings.csv

	user_1	**user_2**	**user_3**	**user_4**	**user_5**
movie_1	1	1	1	1	1
movie_1	2	2	2	2	2
movie_1	3				
movie_1	4				

Table 2: Example of Training Set

	user_1	**user_2**
movie_1	1	1
movie_1	2	2

Table 3: Example of Validation Set

	user_1	**user_2**
movie_1	1	1
movie_1	2	2

Table 4: Example of Testing Set

Please see the file *SplitData.ipynb* for data pre-processing.

3 Model Fitting & Evaluation

3.1 Baseline Popularity Model

The first model we built is the Popularity Baseline Model, which provides a baseline score for the ALS model to compare with. Table 5 shows the performance of applying the popularity baseline model to both *ml-latest-small/ratings.csv* and *ml-latest/ratings.csv* evaluated with Mean Average Precision.

	MAP		NDCG	
	small	full	small	full
Validation	0.000133548	8.1358e-07	0.0018802	9.5618e-06
Testing	7.78695e-05	4.8927e-07	0.0018452	5.7426e-06

Table 5: MAP@100 and NDCG@100 of Popularity Baseline Model

Then, we introduced the dumping variable to decrease the impact of unstable estimates caused by very few interactions. The dumping value we set is 101. It greatly increased the performances of the baseline model shown in Table 6.

	MAP		NDCG	
	small	full	small	full
Validation	0.04787198	0.0268655	0.17190824	0.09703434
Testing	0.04490526	0.02683854	0.1666451	0.09733990

Table 6: MAP@100 and NDCG@100 of Popularity Baseline Model with dumping = 101

Since the number of users in *ml-latest/ratings.csv* is considerably larger than the number of users in *ml-latest-small/ratings.csv*, we increased the dumping value from 101 to 10100 for *ml-latest/ratings.csv*. Table 7 shows the performance with MAP@100 and NDCG@100 with the updated dumping value.

	MAP	NDCG
Validation	0.04150158	0.14115271
Testing	0.04167664	0.14183138

Table 7: MAP@100 and NDCG@100 of Popularity Baseline Model for *ml-latest-small/ratings.csv* with dumping = 10100

Please see *baseline_small.py*, *baseline_fitting.py*, and *baseline_predicting.py* for implementation of the popularity baseline mode.

3.2 Alternating Least Squares Model

We built the ALS model on peel using spark's package, *pyspark.mllib.recommendation.ALS()*, on both *ml-latest-small/ratings.csv* and *ml-latest/ratings.csv*.

In order to optimize the model accuracy with evaluation metrics of MAP@100 and NDCG@100, We decided to tune the following hyperparameters:

- rank (rank of the factorization)
- regParam (regularization parameter)
- alpha (alpha for implicit preference)
- maxIter (max number of iterations)

3.2.1 ALS on Small Dataset

Figures below show the tuning result on the small dataset (*ml-latest-small/ratings.csv*).

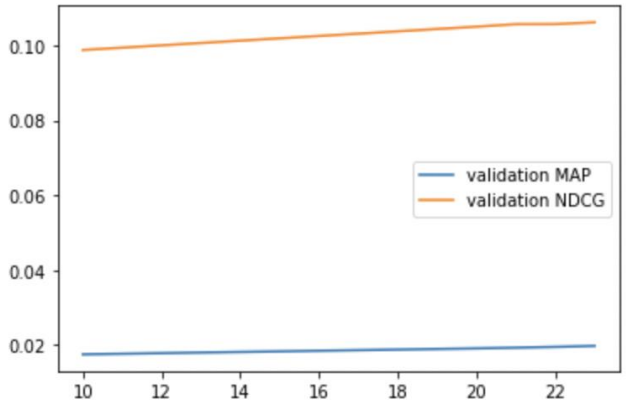

Figure 8: MAP@100 and NDCG@100 of Spark ALS Model for *ml-latest-small/ratings.csv* with different maxIter levels

Figure 8 shows the ALS model performance with different maxIter levels. An increase in maxIter would increase the accuracy of the model. However, this would significantly increase runtime and we decided to stop at maxIter of 23.

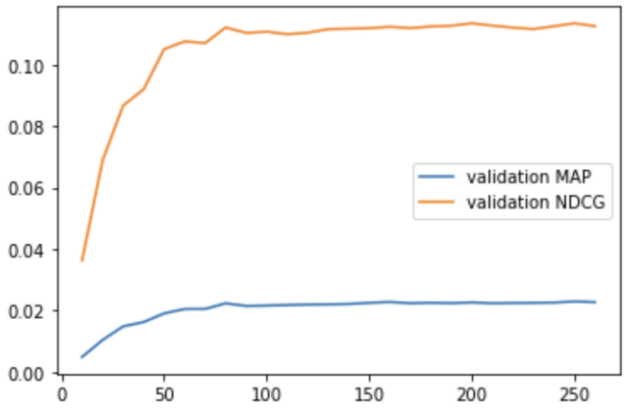

Figure 9: MAP@100 and NDCG@100 of Spark ALS Model for *ml-latest-small/ratings.csv* with different rank levels

Figure 9 shows an increase in both MAP and NDCG from a rank of 10 to 80, and then both evaluation metrics of accuracy remain in a horizontal trend.

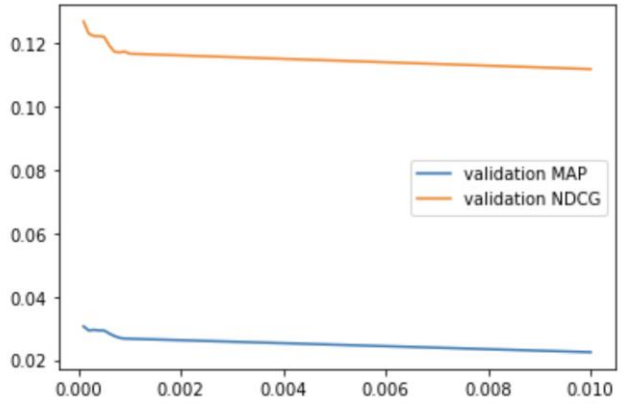

Figure 10: MAP@100 and NDCG@100 of Spark ALS Model for *ml-latest-small/ratings.csv* with different regParam levels

Figure 10 shows a decrease in both MAP and NDCG with increasing regParam levels. And tuning with different alpha levels did not show any effect on either MAP or NDCG performance.

Final result: based on the previous test result, we set the final parameter values for *ml-latest/ratings.csv* using ALS model as following:

- rank = 150
- maxIter = 20
- regParam = 0.0001
- alpha = 5

The resulting MAP and NDCG value for both validation set, and testing set are shown in Table 11.

	MAP	NDCG
Validation	0.030559	0.126921
Testing	0.020754	0.104627

Table 11: MAP@100 & NDCG@100 with Best Parameter for Validation Dataset & Testing Dataset of *ml-latest-small/ratings.csv*

3.2.1 ALS on Full Dataset

On the full dataset (*ml-latest/ratings.csv*), with a significant increase in runtime, we decided to only test the following combinations of hyperparameters shown in Table 11. Although we only carried out a few tests, the results show the same pattern as we've obtained in tests done on the small dataset.

	rank	maxIter	regParam	alpha	val_MAP	val_NDCG
1	100	1	0.01	10	6.5877e-05	5.7348e-04
2	100	10	0.01	10	0.002762	0.017911
3	100	10	0.1	10	0.000313	0.002549
4	100	10	0.01	5	0.002762	0.017911
5	200	10	0.01	5	0.004636	0.030075
6	500	10	0.00001	5	0.002242	0.018196
7	200	15	0.01	5	0.005796	0.038687
8	200	20	0.01	5	0.006807	0.044597

Table 12: MAP@100 and NDCG@100 of Spark ALS Model for *ml-latest-small/ratings.csv* with different hyperparameter combinations

In Table 12, each row represents a test done with each hyperparameter combination and resulting accuracy in the last two columns val_MAP (MAP@100) on validation data, and val_NDCG (NDCG@100) on validation data. From Table 11, we had following conclusion:

- With test 1 and test 2, it is shown that increasing maxIter can indeed increase accuracy in both evaluation metrics, just as proved in tests on the small dataset.
- With test 2 and test 3, it can be concluded that performance of the model increases with a smaller regParam level.
- Between test 2 and test 4, we see that no difference does alpha level make on the performance of the model.
- With test 4 and test 5, it is shown that performance of the model would increase with a larger rank level.

From above conclusions, we noticed that increasing rank level, decreasing regParam, and increasing maxIter can improve the model performance, so we decided to further prove it.

- With test 6, we kept increase rank level and decrease the regParam, however the performance start to decrease.
- With test 7 & 8, we further increased the value of maxIter, and the performance keep increasing. However, after we tried maxIter > 20, the system crashed.

Final result: according to the previous test result, we set the final parameter values as following:

- rank = 200
- maxIter = 20
- regParam = 0.01
- alpha = 5

This set of parameters brings out a reasonably well-trained model, proven by test 8 above, without compromising too much model efficiency with runtime. The resulting MAP and NDCG for both validation set, and testing set are shown in Table 13.

	MAP	NDCG
Validation	0.006807	0.044597
Testing	0.006853	0.044525

Table 13: MAP@100 & NDCG@100 of Best Parameter for Validation Dataset & Testing Dataset of *ml-latest-/ratings.csv*

Please see *ALS_small.py* and *ALS_full.py* for implementation of the ALS model, and *ALS_hyper.py* for hyperparameter tuning

3.3 Single Machine Implementation – LightFM

After building our ALS model in Spark, we looked for ways to improve both efficiency and accuracy of our model by comparing it to a Single Machine Implementation and we decided to train our model with LightFM. LightFM is an implicit feedback recommender, which treats the absence of ratings as users' conscious choices instead of absence of information. When a user decides not to watch a movie, instead of treating the movie rating as unknown, LightFM treats it as valuable information as users pick on what they like and like not to watch, described as the missing-not-at-random phenomenon. It also establishes that by choosing to watch a particular movie, the user has expectations that he / she might like it, despite the actual rating, which happened after the decision process.

We used the same training, testing and validation sets as we did in previous models. Since sizes of our partitioned training and testing datasets are not the same, we had to reassign movie IDs into consecutive integers in order to avoid dimension error. After appending new movie IDs to existing train, test, and validation test sets, we dropped original movieId columns, adjusting unmatched dataset dimensions, as well as timestamp columns, which are irrelevant in model training. Then we built interactions, converting the long data frame into a sparse matrix. We then fit our training data to both WARP and BPR models with different sets of hyperparameters and compared them to our ALS model.

The evaluation metric we decided to use is precision_at_k because the same performance metric is available in Spark ALS, making it easier to compare accuracy between two models. Apart from accuracy, we also compared program runtime between ALS and LightFM as a measure of model efficiency. Results for both model efficiency and accuracy are shown in below table.

		ALS		LightFM (WARP)	
		small	**full**	**small**	**full**
rank = **10** regParam = **0.1** maxIter = **10**	**Precision**	0.022407	0.000166	0.007454	0.006774
	Time	34.0023	413.8259	0.9423	783.8240
rank = **10** regParam = **0.1** maxIter = **1**	**Precision**	0.012361	0.000040	0.006343	0.013792
	Time	30.6126	498.7871	0.542376	596.5954
rank = **100** regParam = **0.01** maxIter = **1**	**Precision**	0.049630	0.000449	0.085231	0.005999
	Time	45.52987	1020.6368	0.7077	1073.1022
rank = **10** regParam = **0.01** maxIter = **10**	**Precision**	0.028241	0.000038	0.095556	0.002873
	Time	31.8018	381.1395	0.8213	1100.6245
rank = **10** regParam = **0.1** maxIter = **20**	**Precision**	0.023102	0.000237	0.006667	0.004513
	Time	34.5263	1274.2124	1.3313	1052.7533
rank = **100** regParam = **0.1** maxIter = 1	**Precision**	0.034167	0.000146	0.008102	0.008705
	Time	30.5141	1738.8394	0.8371	1052.9542

Table 14: Runtime and Precision@100 of Spark ALS and LightFM models

While training, we found that the BRP model, Bayesian Personalised Ranking, optimizes ROC AUC, whereas, The WARP model, Weighted Approximate-Rank, optimizes the top of the recommendation list (precision@k). Therefore, we decided to only include WARP results in the above table as BRP gives less desirable results when we measure accuracy in terms of precision.

From above table, we can conclude that compared with ALS, less runtime is needed for the smaller dataset in the LightFM model and more runtime is required in the LightFM model for the entire dataset. In most cases, precision is higher with the LightFM model comparing with ALS.

We also used hyperparameter tuning in order to achieve higher efficiency as well as accuracy for the LightFM model. As regParam gets smaller, higher accuracy is achieved in the full dataset, while lower accuracy is obtained in the small dataset. Efficiency slightly increases with a shorter runtime on the full dataset as regParam gets larger. As maxIter increases, accuracy did not improve as we expected, whereas efficiency decreases where runtime increases on both small and full datasets. In addition, as rank gets larger, accuracy did not improve as we expected and efficiency worsens with a much longer runtime.

Final result: according to the previous test result, we set the final parameter values for LightFM as following:

- rank = 10
- maxIter = 1
- regParam = 0.1

This set of parameters brings out a reasonably well-trained model, proven by the second test in the table above, without compromising too much model efficiency with runtime.

In conclusion, higher accuracy is achieved in LightFM model, comparing with ALS, by incorporating missing movie ratings as a feature during users' movie selection process.

Please see *LightFM_small.py* and *LightFM_full.py* for implementation of the Single Machine Implementation LightFM model.

3.3 Fast Search

The user factors and items factors generated by the ALS model could help us to establish spatial data structure, which could accelerate query time. In the second extension, we've implemented a fast search algorithm using Annoy and also established a brute force algorithm for efficiency gain comparison. Annoy is a package that is specifically used to approximate k nearest neighbors. Instead of computing all inner products between the user factor U[I] and item factors V[j], Annoy allows us to index the data, creating a spatial data structure which accelerates query search time.

To begin with, we have first taken the users factors and items factors from our best performing ALS model. Then, the first find k nearest neighbor algorithm we implemented is Brute force. In this project, the numbers of neighbors we used for analysis is 100 neighbors. The time efficiency and evaluation are recorded in Table 17. Then we implemented the fast search algorithm using Annoy. The algorithms iterate through all the user queries in the small and full dataset, and the average query time and average query accuracy are being measured. Queries accuracy is calculated by finding the percentage of how many outputs of each algorithm exist in the actual movie list for each query.

The parameter which needs to be tuned for this algorithm is n_tree (the number of trees in this structure). After conducting trials with n_tree = [10,100,200,500,700,1000], the tradeoff between accuracy and time efficiency is best at n_tree=100 for small dataset and n_tree = 200 for full dataset. The results for average query time and accuracy produced by each chosen parameter are attached below.

	params	Times	Accuracy
1	10	0.000035	0.016656
2	100	0.000062	0.019754
3	200	0.000095	0.019082
4	500	0.000167	0.013574
5	700	0.000223	0.013557
6	1000	0.000342	0.013607

Table 15: Fast Search performance with different n_tree parameter for small dataset

	params	Times	Accuracy
1	10	0.000049	0.001678
2	100	0.000068	0.001866
3	200	0.000106	0.005166
4	500	0.000204	0.005164
5	700	0.000297	0.005164
6	1000	0.000467 I	0.005164

Table 16: Fast Search performance with different n_tree parameter for full dataset

Comparing the best performing fast search algorithm with the brute force algorithm, we get the results in Table 17. As shown by the following result, fast search is much faster than Brute force in terms of query time and actually produce better approximations than Brute force algorithms as well: it is approximately 10 times faster for small dataset, 50 times faster for 10000 users and can process about 270,000 users queries at a considerably short time when brute force algorithm is too computationally expensive to be computed. In conclusion, indexing data to create spatial data structure would significanty improve time efficiency of query search.

	Algorithm	Time taken	Accuracy
Small Dataset	Brute Force	0.000566	0.016394
	Fast Search	0.000057	0.019754
Full Dataset (sample 10000 users)	Brute Force	0.004414	0.002620
	Fast Search	0.000089	0.005133
Full Dataset	Brute Force	Too expensive to be computed	Too expensive to be computed
	Fast Search	0.000108	0.005166

Table 17: Brute force VS. Fast search comparison

Please view *full_Fast_Search_vs_Brute_Search.ipynb* and *small_Fast_Search_vs_Brute_Search.ipynb* for implementation of Brute force and fast search algorithm. Note: please read readme.md before running the Jupiter notebook

CONTRIBUTION

- Cooper Zhao:
  - Data Pre-processing
  - Baseline Popularity Model
  - ALS Model
  - Hyper-parameter tuning on ALS model

- Vanessa Xu:
  - Extension 1: Single Machine Implementation (LightFm)
  - Hyper-parameter tuning on ALS & LightFM model

- Catheriane Zheng:
  - Extension 2: Fast Search
  - Hyper-parameter tuning on ALS model

REFERENCES

[1] Harper, F. Maxwell, and Joseph A. Konstan. "The Movielens Datasets." *ACM Transactions on Interactive Intelligent Systems*, vol. 5, no. 4, Jan. 2016, pp. 1–19, doi:10.1145/2827872.

[2] *RankingEvaluator — PySpark 3.1.1 Documentation.* https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.ml.evaluation.RankingEvaluator.html. Accessed 16 May 2022.

[3] *ALS — PySpark 3.2.1 Documentation.* https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.recommendation.ALS.html#pyspark.ml.recommendation.ALS.rank. Accessed 15 May 2022.

[4] Kula, Maciej. "Metadata Embeddings for User and Item Cold-Start Recommendations." *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.*, edited by Toine Bogers and Marijn Koolen, CEUR-WS.org, 2015, pp. 14–21, http://ceur-ws.org/Vol-1448/paper4.pdf. Accessed 17 May 2022.