Recommendation System on the MovieLens Dataset[1]

Spring 2021 DS-GA 1004 Final Project

Luoyao Chen
Center for Data Science
New York University
lc4866@nyu.edu

Shaojun Jiang
Center for Data Science
New York University
sj2539@nyu.edu

Yue Feng
Center for Data Science
New York University
yf1451@nyu.edu

1 Introduction

Recommendation plays an important role for movie websites. The algorithms for recommendation systems include popularity-based recommendation, collaborative filtering, as well as content-based filtering. The popularity-based recommendation provides recommendations based on how "popular" an item is (e.g. the more ratings a movie gets, the more popular the movie is). Collaborative filtering factorizes a utility matrix into a user matrix and an item matrix hence provides recommendations based on how similar a new user is to the user in the database. Content-based filtering allows a new user to be categorized by some new features, and the latter is mapped to the latent feature space. In this project, using the MovieLens dataset, we constructed a popularity-based model (using the top 100 most popular movies) and a ALS-based collaborative filtering, using PySpark 3.0.1 (and 3.2.1 for LightFM). In addition, we investigated some performance improvements (LightFM), and visualized the learned item representations using UMAP.[1]

2 Dataset and Preprocessing

2.1 MovieLens Dataset

We used the MovieLens dataset[1], which contained two versions (small and large). Specifically, the small version contained 9,000 movies by 600 users (files: links.csv, movies.csv, ratings.csv, tags.csv), and the large version contained 58,000 movies by 280,000 users (files: links-large.csv, movies-large.csv, ratings-large.csv, tags-large.csv, and genome-scores-large.csv).

2.2 Preprocessing

To deliver meaningful and relatively accurate recommendations, we performed data preprocessing before implementing the algorithms. We first searched for duplicate records in the datasets and kept only the first record. Then, we dropped ratings with null values, lower than 0.5, or higher than 5. In addition, we only kept movies with more than 10 ratings and users with more than 10 rating records.

After performing the above operations, we split our data into train/validation/test sets as dataframes based on userId with weights 0.6, 0.2, 0.2 respectively. Then, to guarantee that we have all the user history in the training set, i.e. to avoid the cold start problem, we sorted each user's rating records in the validation and test sets using timestamps and appended the earlier half of the rating records from each user onto the training set, while leaving only the latter half of the ratings in the validation and test sets. Finally, we repartitioned each of the train/validation/test data frames and created parquets for further operations.

3 Choice of Evaluation Criteria

In this project, we used two metrics to evaluate the performance of our models, precision and mean average precision(MAP).

First of all, we chose to use precision as our evaluation metric. Since we wanted to implement a recommendation system that provided a ranking list to each user, we were interested in the accuracy of the top items recommended. As such, we chose precision at 100 as our evaluation metric:

$$\frac{\#\ within\ top\ k\ recommendations\ that\ match\ user\ preference}{\#\ of\ items\ that\ are\ recommended}$$

Also, we used mean average precision:

$$MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q}$$

[1] Our implementation details can be found on: https://github.com/nyu-big-data/final-project-group_3

While precision considers only a subset of recommendations, mean average precision rewards us for placing the correct recommendations on the top of the list. This metric evaluated the model even more comprehensively, since a user has a finite amount of time and attention, not only do we want to know what kinds of products they might choose, but also which are chosen the most or which we are the most confident of for recommendation.

4 Model Implementations

4.1 Baseline Model

A popularity-based model uses the items that are the most popular for recommendation. For example, if a product has previously been consumed by more users, then it will have a higher chance to be liked by a new user..

To implement the model, for each movie appearing in the training set, we calculated an average rating, and to prioritize movies with more rating records than those with fewer, we also added a damping factor into the denominator. We sorted the movies by averaging ratings and chose the top 100 to recommend to users. For tuning, we compared damping factors on the validation dataset (damping_factor= [0, 0.01, 0.1, 1, 10].)

Table1: Baseline Results on Validation Set

Damping Factor	Small Dataset	Large Dataset
0	0.000357	0.000004
0.01	0.000164	0.000005
0.1	0.000242	0.000005
1	0.036393	0.022277
10	**0.041498**	**0.028946**

Figure1: Baseline Results on Validation Set

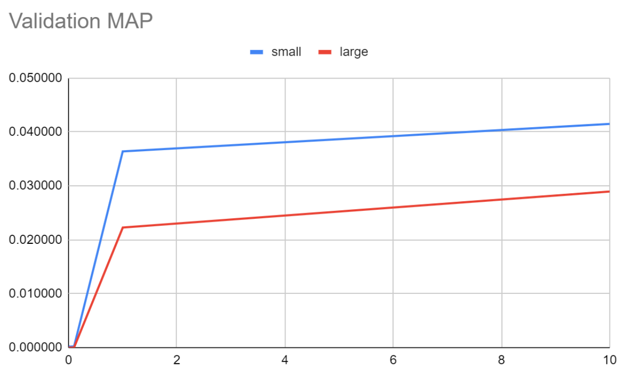


Table2: Test result with/out damping factor

	damping factor	small	large
test MAP	0	0.003286	0.000004
test MAP	10	**0.039714**	**0.021160**

After trying different damping factors, we found that the model with a damping factor = 10 had the best performance on both small and large datasets, as it greatly improved the MAP compared to the model without a damping factor (MAP increased from 0.039714 to 0.021160).

Also, the result showed that the popularity-based model had a better performance on the small dataset compared to the large, which could be caused by the sparsity of the large dataset. In the large dataset, most of the movies got few or even no rating data points, thus it would be harder to find out the general trend and our selection of the highly-rated movies was biased.

4.2 Latent Factor Model

A collaborative filtering model estimates the utility matrix R using the product of two lower-rank matrices, U(user) and V(item). Specifically, such estimation was based on Alternative Least Square (ALS).

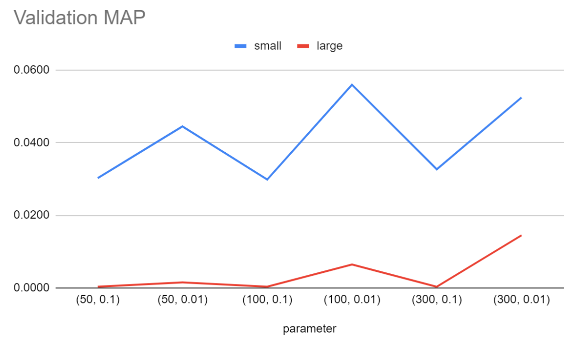
$$R = UV^T$$

To implement the model, we used the pyspark.ml.recommendation package to build a ALS model, and we tuned different parameters rank (the number of latent factors) and regularization factors (penalty coefficient) to find out the best parameter combination. Specifically, we tried rank = [50, 100, 300], and reg = [0.01, 0.1].

Table 3: ALS model on Validation Set

rank	reg	Small validation MAP	Large validation MAP
50	0.1	0.0302	0.0004
50	0.01	0.0445	0.0015
100	0.1	0.0298	0.0004
100	0.01	**0.0559**	0.0065
300	0.1	0.0327	0.0003
300	0.01	0.0524	**0.0145**

Figure2: ALS model on Validation Set



Based on the results above, we concluded that for the small dataset, the model with rank = 100 and reg = 0.01 performed the best (MAP = 0.054642); While for the large dataset, the model with rank = 300 and reg = 0.01 performed the best, (MAP = 0.014537). The result indicates that using a larger rank and a smaller regularization factor helped improve the MAP.

4.3 Baseline vs Latent Factor Model

Compared to the baseline model, the ALS model performed better on the small dataset, while the baseline model performed better on the large dataset. Even so, the MAP scores of the two models on the large dataset were both lower than on the small. This could be explained by the algorithms the two models used, the ALS model used matrix factorization to find out two latent factor matrices. It would be easier to find the matrix using a small dense dataset, while the sparse large dataset would be hard to decompose into the user and item matrix. Thus, for a small dataset, the ALS model had better performance compared to the baseline model, but when the rank of the latent factors increased, the prediction of ALS became less reliable and the model performance would be even worse than the baseline model.

5 Documentation of Extension(s)

5.1 Extension 1: Comparison to single-machine implementations

5.1.1 A single-machine implementation — LightFM model

LightFM[2] is a python implementation of recommendation algorithms that can be run on a single machine as opposed to Spark which is primarily designed for parallel computing. We implemented this extension of the project to investigate the differences in efficiency and accuracy between the LightFM and ALS algorithms.

To run LightFM, we followed the official documentation of LightFM version 1.16 to construct the desired format of dataset from our datasets and build the recommendation model. Since LightFM is a single-machine implementation and requires installing the LightFM package, we ran the program in a Singularity container on the NYU Greene cluster. We set up the environment with a chosen Singularity image[2] and pip installed all the required packages, including LightFM, Pandas, Pyarrow, and Fastparquet, on the Singularity container for LightFM. To execute the program, we submitted a slurm job for each run, requesting only 1 node and 1 CPU per task to ensure it was a single-machine implementation. In addition, we also set the num_thread hyperparameter to 1 in LightFM to ensure the model is run by one single machine.

Corresponding to Spark ALS algorithm's hyperparameters rank of the matrix factorization model (rank) and regularization parameter (regParam), LightFM has no_components (the dimensionality of the feature latent embeddings), item_alpha, and user_alpha. We ran our LightFM model on both the small and large datasets with different combinations of rank and regParam as hyperparameter tuning for later comparison with Spark ALS.

Table 4: LightFM model on Validation Set

no_components	user_alpha= item_alpha	small validation precision at 100	large validation precision at 100
30	0.1	0.004844	0.007273
30	0.01	**0.042578**	0.020426
50	0.1	0.003672	0.008916
50	0.01	0.012500	(diverged)
100	0.1	0.004297	**0.021736**
100	0.01	0.003281	0.006222
300	0.1	0.003906	(diverged)
300	0.01	0.002187	(diverged)

As we could see from the above tables, the best hyperparameters for LightFM on the small dataset are no_components = 30, user_alpha = item_alpha = 0.01, and the model precision at 100 was 0.042578; on the large dataset are no_components = 100, user_alpha = item_alpha = 0.1, and the model precision at 100 was 0.021736. Note that due to the large size of the dataset and the computation implemented by LightFM, some combinations of the hyperparameters led to divergence.

[2] The Singularity image we used is: /scratch/work/public/singularity/cuda11.2.2-cudnn8-devel-ubuntu20.04.sif. We follow the setup posted on NYU HPC documentation: https://sites.google.com/nyu.edu/nyu-hpc/hpc-systems/greene/software/singularity-with-miniconda?authuser=0

5.1.2 ALS and LightFM Model Comparison

To compare the efficiency and accuracy of LightFM and Spark ALS fairly, we compared the precision at 100 on the test set of the best-performing LightFM and Spark ALS models trained on different sizes of training data. Again, in order to import the LightFM package to our program, we set up a Singularity container using the same Singularity image as mentioned in *5.1.1*, except that we pip install an extra Pyspark package to run ALS. We randomly selected a subset of the original training set with size 20%, 50%, 80%, and 100% of the original training set size (both small and large) respectively for the performance comparison. The results can be seen in the tables below.

Table 5: ALS model and LightFM model comparison on subsets of the small dataset

Training subset proportion	ALS time	LightFM time	LightFM time/ALS time ratio	ALS precision	LightFM precision
20%	31.127489	**0.020936**	0.000673	0.040588	**0.050450**
50%	33.552058	**0.058002**	0.001729	0.041111	**0.071696**
80%	39.001965	**0.095329**	0.002444	0.049130	**0.081071**
100%	42.968802	**0.125360**	0.002917	0.057766	**0.078839**

Table 6: ALS model and LightFM model comparison on subsets of the large dataset

Training subset proportion	ALS time	LightFM time	LightFM time/ALS time ratio	ALS precision	LightFM precision
20%	2130.607455	**5.785755**	0.002716	**0.037951**	0.034508
50%	4040.117715	**18.571602**	0.004597	0.030861	**0.051424**
80%	5771.094034	**29.389645**	0.005093	0.027738	**0.050025**
100%	6897.781200	**40.277010**	0.005839	0.027220	**0.057242**

As we can see in Table 5 and Table 6, LightFM is much more efficient than ALS on subsets of both the small and large datasets - the runtime of LightFM is 0.6% to 5.8% of that of ALS. In terms of accuracy, based on our chosen evaluation metric and training set sizes, LightFM is about 24.3% to 74.4% better than ALS on the subsets of the small dataset. On the subsets of the large dataset, for all subset proportions except for 20%, LightFM outperforms ALS by 66.6% to 110.2%.

5.2 Qualitative error analysis

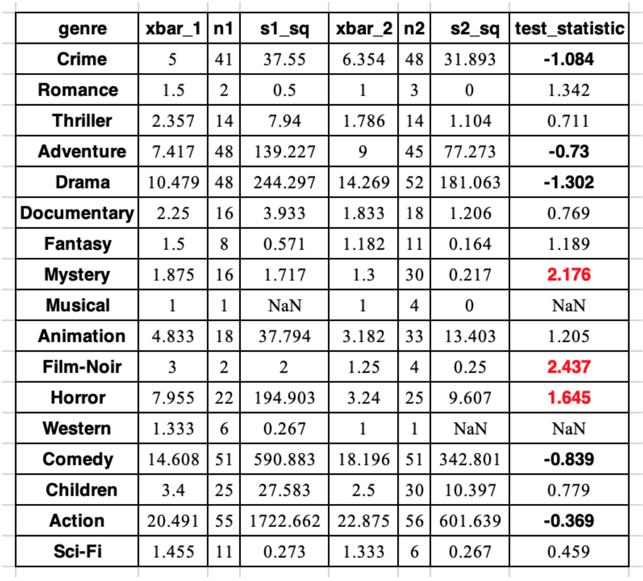
5.2.1 Investigate the trend of users who produce low scoring predictions

Using the training set, we fitted the ALS model and obtained the predictions for the mean of ratings and their corresponding mean confidence. We used the average confidence of each user as the criterion to distinguish low-prediction-score from high-prediction-score users. Specifically, we divided the users from the test set into high and low confidence user groups, according to the median average confidence. For each group, we obtained the movie genres using movieId from the movies dataset. For each genre (19 in total), we conducted a cross-group t-test. It was worthwhile to mention that the distribution of the high vs low confidence group were different, which motivated us to use a different criteria. We first used 0.5 as the threshold of confidence level, which gave very imbalance counts in low vs high confidence groups (caused issues when calculating the t-statistic), and then we decided to use the median, which provided a more reasonable division.

Here, xbar_1, n1 and s1_sq represented the low confidence users, while n2 xbar_2, s2_sq represented high confidence users.

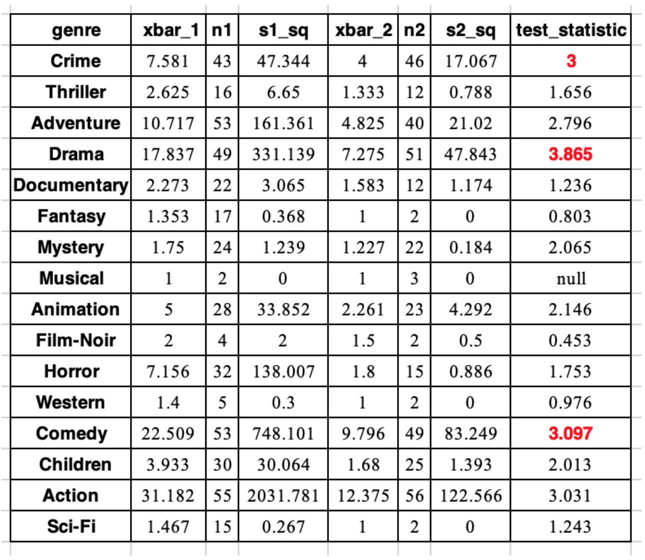
Test statistics = $\frac{\{x1-x2\}}{sp\sqrt{\frac{1}{n1}+\frac{1}{n2}}}$, $s_p = \sqrt{\frac{(n1-1)s_1^2+(n2-1)s_2^2}{n1+n2-2}}$

Table 7: The t-test table using best ALS model on the small dataset (rank = 100, regularization = 0.01)



genre	xbar_1	n1	s1_sq	xbar_2	n2	s2_sq	test_statistic
Crime	5	41	37.55	6.354	48	31.893	**-1.084**
Romance	1.5	2	0.5	1	3	0	1.342
Thriller	2.357	14	7.94	1.786	14	1.104	0.711
Adventure	7.417	48	139.227	9	45	77.273	**-0.73**
Drama	10.479	48	244.297	14.269	52	181.063	**-1.302**
Documentary	2.25	16	3.933	1.833	18	1.206	0.769
Fantasy	1.5	8	0.571	1.182	11	0.164	1.189
Mystery	1.875	16	1.717	1.3	30	0.217	**2.176**
Musical	1	1	NaN	1	4	0	NaN
Animation	4.833	18	37.794	3.182	33	13.403	1.205
Film-Noir	3	2	2	1.25	4	0.25	**2.437**
Horror	7.955	22	194.903	3.24	25	9.607	**1.645**
Western	1.333	6	0.267	1	1	NaN	NaN
Comedy	14.608	51	590.883	18.196	51	342.801	**-0.839**
Children	3.4	25	27.583	2.5	30	10.397	0.779
Action	20.491	55	1722.662	22.875	56	601.639	**-0.369**
Sci-Fi	1.455	11	0.273	1.333	6	0.267	0.459

Table 8: The t-test table using the best ALS model on the large dataset (rank=300, regularization = 0.01)



genre	xbar_1	n1	s1_sq	xbar_2	n2	s2_sq	test_statistic
Crime	7.581	43	47.344	4	46	17.067	**3**
Thriller	2.625	16	6.65	1.333	12	0.788	1.656
Adventure	10.717	53	161.361	4.825	40	21.02	2.796
Drama	17.837	49	331.139	7.275	51	47.843	**3.865**
Documentary	2.273	22	3.065	1.583	12	1.174	1.236
Fantasy	1.353	17	0.368	1	2	0	0.803
Mystery	1.75	24	1.239	1.227	22	0.184	2.065
Musical	1	2	0	1	3	0	null
Animation	5	28	33.852	2.261	23	4.292	2.146
Film-Noir	2	4	2	1.5	2	0.5	0.453
Horror	7.156	32	138.007	1.8	15	0.886	1.753
Western	1.4	5	0.3	1	2	0	0.976
Comedy	22.509	53	748.101	9.796	49	83.249	**3.097**
Children	3.933	30	30.064	1.68	25	1.393	2.013
Action	31.182	55	2031.781	12.375	56	122.566	3.031
Sci-Fi	1.467	15	0.267	1	2	0	1.243

In conclusion, on the small dataset, low-confidence users paid more attention to genres such as Mystery, Film-Noir, Horror (test-statistics > 1.64). On the large dataset, low-confidence users paid more attention to genres such as Crime, Drama, Comedy, etc., compared with high-confidence users. However, it was noticeable that some genres were more "unstable", especially as the size of the dataset increased, such as Crime, Adventure, Drama, Comedy and Action. We concluded that those 5 genres were the most popular genres, hence recommending precisely in those 5 genres were more difficult in general.

5.2.2 Visualize Learned Item Representation

A UMAP works by reducing the dimension of the item features. Namely, we first factorized the utility matrix into user and item matrix, each with a fixed number of latent features (100 latent features on the small dataset and 300 features on the large). Using n_neighbourhood = 15, we reduced the number of latent features to 2 to visualize the item representation. We labeled the movie by their genres, and visualized them below.

Figure3: Small dataset, rank = 100, reg =0.01

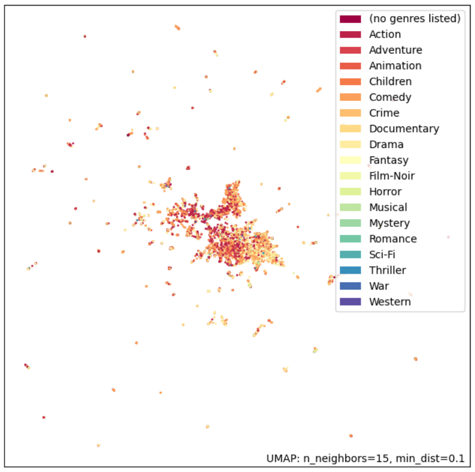
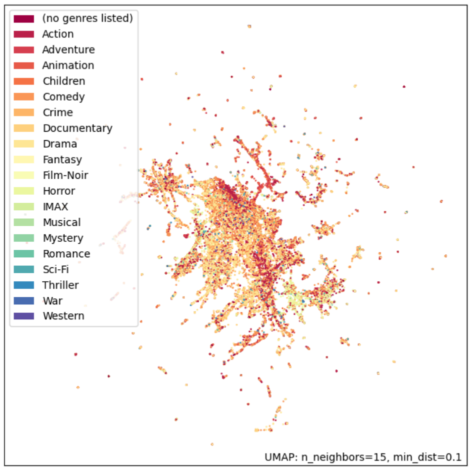


Figure4: large dataset, rank = 300, reg =0.01



As can be seen from the plots, using only 2 dominant latent features, movie genres could not be clearly separated; but we can still conclude that the most popular genres were Action, Adventure, Animation, and Drama.

6 Conclusion

In this project, we investigated two types of recommendation models, each with an advanced version (4 in total): popularity-based mode, with and without damping factors, and collaborative filtering recommendation model using ALS or LightFM algorithm.

To compare within each type of model: For the popularity-based model, adding a damping factor greatly improved the model performance (increased MAP from 0.003286 to 0.039714 on the small dataset, and increased MAP from 0.000004 to 0.021160 on the large dataset); the ALS model and LightFM model both needed hyperparameter tuning to achieve better performance, and both performed better on the small dataset than on the large. In addition, LightFM model was both more efficient and accurate than the ALS model on both the small and large datasets.

To compare across two types of models: When comparing the popularity-based model with the ALS model, we found that the ALS model gave the best performance on the small dataset, while the baseline model with damping factor =10 provided the best performance on the large dataset.

7 Collaboration & Contribution

- Data Cleaning, Pre-Processing: Yue Feng
- Baseline Model Building and Tuning: Yue Feng
- ALS Model Building and Tuning: Yue Feng, Luoyao Chen
- Extension(LightFM) Building and Tuning: Shaojun Jiang
- Extension(Quantitative Error Analysis) Building and Tuning: Luoyao Chen
- Report: Yue Feng, Shaojun Jiang, Luoyao Chen

8 Files Description

- Data Cleaning and Preprocessing
    - data_train_test_split.py
- Baseline Model:
    - baseline_new.py
- ALS Model:
    - ALS_model_map.py
- LightFM
    - comparison.py
    - extension1.py
    - run_lfm.slurm
    - run_comparison.slurm
- Quantitative Error Analysis
    - extension4.py

9 REFERENCES

[1] F. Maxwell Harper and Joseph A. Konstan. 2015. *The MovieLens Datasets: History and Context.* ACM Trans. Interact. Intell. Syst. 5, 4, Article 19 (January 2016), 19 pages. https://doi.org/10.1145/2827872

[2] Maciej Kula. 2015. *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. (CEUR Workshop Proceedings), CEUR-WS.org, 14–21. Retrieved from http://ceur-ws.org/Vol-1448/paper4.pdf