MovieLens Recommender System with Explicit Feedback

Yiqi Yuan
yy3754@nyu.edu

Yunxin Gao
yg1434@nyu.edu

Jiawen Fan
jf3539@nyu.edu

1. Introduction

With the rise of e-commerce, online videos, and other web services in the last few decades, the recommender system has taken a significant place in our daily lives. In our final project, we trained and compared different algorithms for recommending movies with explicit feedback--a popularity baseline model, two latent factor models (ALS, LightFM), and performed qualitative error analysis on the resulting latent factors. Due to the enormous scale of the data and the high complexity of the algorithm, we took advantage of Peels (NYU HPC) to conduct model training, testing, and evaluation.

2. Data Processing

2.1 Introduction to MovieLens datasets

We used the latest version of MovieLens datasets which could be retrieved from MovieLens Latest Datasets | GroupLens. The datasets describe 5-star ratings and free-text tagging from MovieLens, a movie recommendation service. A "small" set and a "full" set are included, and we performed analysis and test on the "small" set first, then adapted analysis on the "full" set for the final report of our recommender system.

2.2 Data Preprocessing

We mainly used Spark and HDFS to conduct data preprocessing steps and data storage. Considering the significance of efficiency, we converted all CSV data files into parquet in the first step and saved the converted Parquet files into HDFS (Hadoop Distributed File System). Column types are also defined for consistency.

For further analysis of the model accuracy of our recommender system, we had to partition data into three parts: training, validation, and testing. All interactions with a rating of 0 are filtered out since they do not provide valuable information.

We first splitted data into three sets based on the number of unique users who rate at least 20 movies in a 6:2:2 ratio. Moreover, in order to make sure that the user who appears in the validation and testing set would also appear in the training set with some ratings, we further splitted both the current validation and testing set by their row numbers, and merged those with even row numbers into the training set. Those with odds row numbers were kept in the original set. Finalized training, validation, and testing sets are generated at this stage, and they are saved as Parquet files into HDFS.

3. Evaluation Metrics

In order to evaluate the performance of our recommender systems, we used the following evaluation metrics:

- **MeanAP:**
  Mean Average Precision at K gives insight into how relevant the list of recommended items is
- **Ndcg:**
  Normalized discounted cumulative gain weights each relevance score based on its position with a normalization factor in the denominator
- **RMSE:**
  Root Mean Squared Error calculates the square root of the absolute value of the residual squared
- **Precision at k:**
  Precision at k is the proportion of recommended items in the top-k set that are relevant

We mainly focused on mean average precision at k in the following analysis.

4. Baseline Popularity

Before implementing a sophisticated model, we began with a popularity baseline model that predicts the same result for every user.

Considering the situation that when a movie gains only one feedback and its rating is five, it would be highly biased if we assume this movie will deserve a rating of five for the whole population. Therefore, we introduced the popularity baseline model with IMDB weighted rating.

To calculate the popularity of the movies with weighted ratings, we utilized the following formula (Seeda, 2021):

$$WR = \frac{v}{v+m} * R + \frac{m}{v+m} * c$$

where

- **v** is the count of ratings of each movie
- **m** is the minimum number of ratings required for each movie (defined by > percentile 70 of total votes)
- **R** is the average rating of each movie
- **c** is the average rating of all the records in the dataset

After calculating each movie's weighted rating, we ordered their rating in descending order. The top 100 movies after order are the recommended movies for all users in our popularity baseline model.

Results:

Dataset	**MAP**	**ndcg(100)**
Small	0.03095	0.12752
Full	0.01185	0.08509

5. Latent Factor Model

Latent Factor Model is a popular collaborative filtering algorithm that attempts to estimate the rating matrix R as the product of two lower rank-matrix. These approximated matrices are called 'latent factor' matrices. Latent factors are the features in the lower dimension latent space projected from the user-item interaction matrix. The idea behind matrix factorization is to use latent factors to represent user preferences or movie topics in a much lower dimension space. In this way, the Latent Factor Model learns to offer personalized recommendations for the users.

$$\tilde{r}_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$

Rating of item i given by user u can be expressed as a dot product of the user latent vector and the item latent vector.[1]

1 https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1

In this project, we aim to optimize the latent model by using an iterative process--ALS: During each iteration, one of our factor matrices will be held constant, while the other will be solved by using least squares. The newly-formed factor matrix will then be fixed while our algorithm is solving the other factor matrix.

5.1. Regularization Parameter:

As the evaluation results shown in Figure 1, we found out that the model has better performance when the reg_para is between 0.01 and 0.05. With the reg_para increasing, the running time of the model keeps decreasing. Based on this exploration, we determined reg_para = 0.05 for the optimal model.

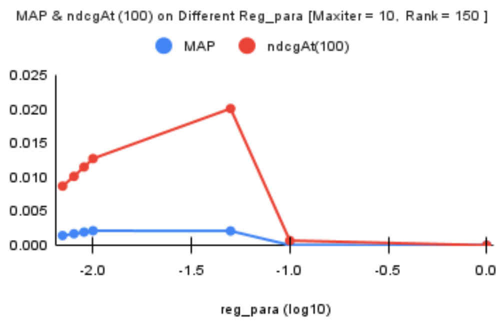

Figure 1:MAP & ndcg on Different reg_para

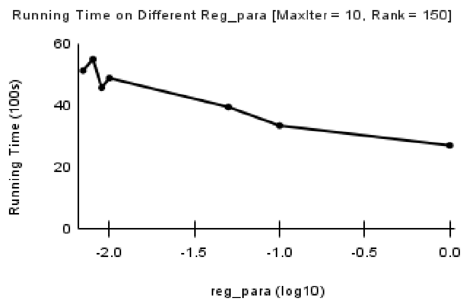

Figure 2: Running Time on Different reg_para

5.2. Ranks of Latent Factor:

Throughout the tuning results, we found that the MAP and ndcgAt (100) increase by increasing rank when the rank is smaller than 300, however, the model performance of Rank = 500 is worse than the metric result of Rank = 300. Because of the limited sources for tuning, we speculated that the model with Rank between 250 to 350 probably had better performance. The running time as shown in figure x significantly increases when Rank is greater than 300. Considering the running time cost, we decided to choose Rank = 250 for our optimal model.

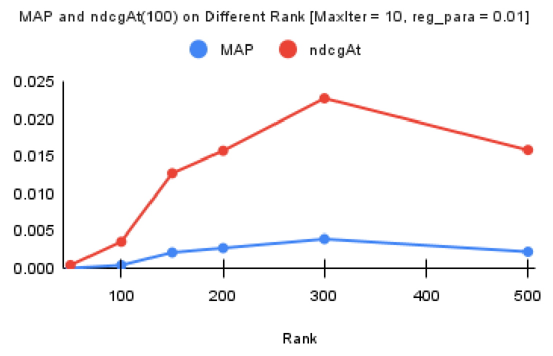

Figure 3: MAP & ndcg on Different Rank

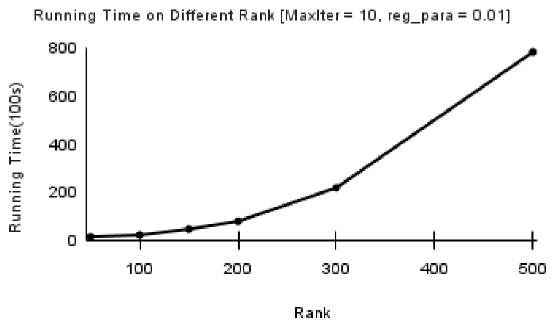

Figure 4: Running Time on Different Rank

5.3. Max Iterations:

We plotted the performance of two sets of tuning in Figure 5 below. From Figure 5, we found out that the performance became better with the increasing MaxIter. We also utilized the RMSE as an evaluation metric here. When MaxIter is greater than 15, RMSE keeps decreasing with the increase of MaxIter. Taking all metrics results into consideration, we believed the model would perform better if MaxIter is greater than 25. Nevertheless, due to limited sources of tuning, we would keep MaxIter = 25 here for further improvement.

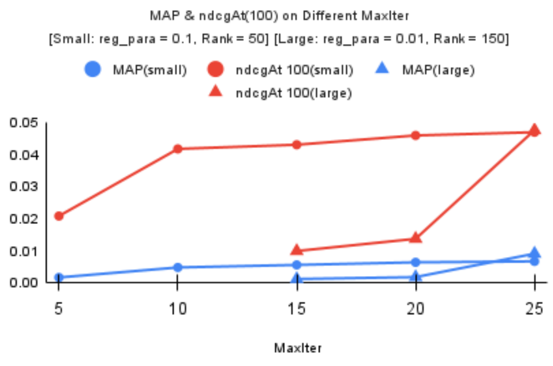

Figure 5: MAP & ndcg on Different MaxIter

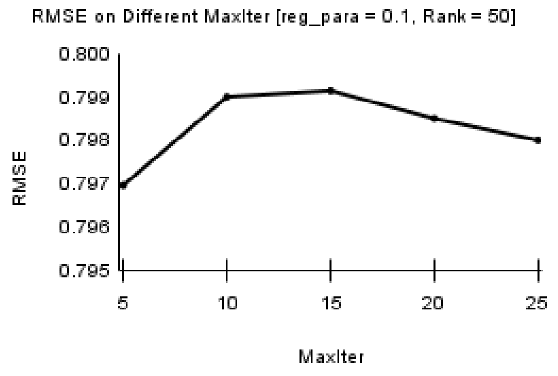

Figure 6: Running Time on Different MaxIter

We finally determined our optimal model with the following hyperparameters:

reg_para	MaxIter	Rank	MAP	ndcg(100)
0.05	25	250	0.00863	0.05617

6. Extension 1: Single Machine(LightFM)

LightFM is a python implementation of recommendation algorithms for both implicit and explicit feedback. Relatively fast(via multithreaded model estimation) and easy to implement, lightFM also produces astonishing-quality results. To align with the ALS model we implemented on Hadoop, we choose to implement lightFM for only explicit recommendation (e.g. rating as the weights) and warm start prediction. Unlike the ALS, lightFM deploys stochastic gradient descent to learn the latent factor embeddings.

Before proceeding to build the model, we first inspect the data and make sure that the usersId and movieId in the validation set and testing set are fully covered in the training set. Due to limited computation resources on a single machine, we have downsized the full dataset into 10% and 25% and compared the lightFM's performance as well as the efficiency of the small (1%), and medium (25%) datasets.

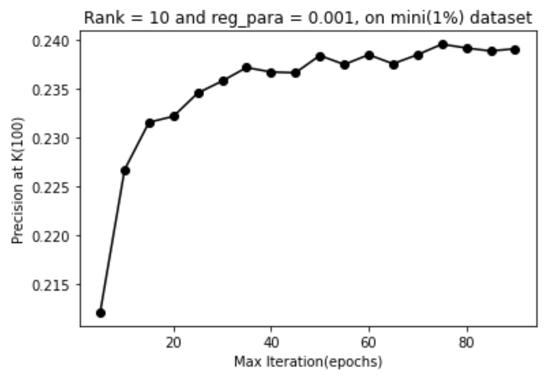

Figure 7: Number of epoch's effects on precisionK

- **lightfm.data.Dataset()**
  We first utilized the built-in method to transform our training, validation, and test data into the format that lightFM accepts.
- **build_interactions(data)**

  This function transformed the built dataset into two COO matrices: the interactions matrix and the corresponding weights matrix.

- **mapping()**

  In order to evaluate lightFM performance on the validation and test dataset, we need to transform our validation and testing data into a sparse coordinate matrix.

- **LightFM()**
  loss: we utilized 'warp', which is the weighted approximate- rank pairwise loss.[2]
  epochs: This is equivalent to the max iterations in ALS
  no_components: similar to the ranks
  user_alpha: regularization parameter

We ran lightfm model (warp) on the small (1%), and medium (25%) with the same hyperparameter combinations as in Spark ALS. Here are some thoughts on the hyperparameter tuning:

1. LightFM performed better on the smaller dataset as well as the efficiency measured by the running time. This is potentially due to different data handling methods, HPC cluster capacity, and inefficient coding in the ALS model.
2. The optimal hyperparameter combinations for Spark ALS and LightFM do not agree. This is partially due to the limited computation capacity on the HPC cluster, which highly limited our choices of the max iterations and ranks for ALS.
3. In theory, ALS could easily ignore the missing entries(unlike lightFM SGD), but it often leads to faster convergence. However, we did not observe this effect on the small and medium datasets. We argued that this might be due to the data size being relatively small and ALS being extremely efficient for handling large-scale datasets. Further investigation could be partitioning larger portions of the data, and testing ALS/LightFM's performance on it.
4. As we can see LightFM will converge at around 25, and we fixed it while tunning other parameters. We have attached the relatively optimal parameters (given time and hpc capacity) of both ALS and LightFM.

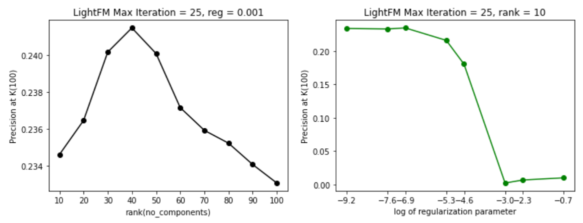

Figure 8: LightFM performance with varying rank and reg_para

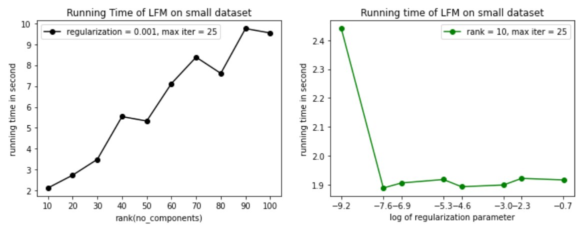

Figure 9: LightFM running time with varying rank and reg_para

Below are the results of the optimal parameter testing on our test dataset.

[2] https://making.lyst.com/lightfm/docs/lightfm.html

Comparison test result between ALS and LightFM optimal parameter on mini dataset(1%)

Model	Max iter	Rank	Reg_para	PrecisionK (100)	Run time
ALS	15	50	0.01	0.0384	54s
Light FM	25	10	0.001	0.21460	2.12s

Comparison between ALS and LightFM on optimal medium test dataset(25%)

Model	Max iter	Rank	reg_Para	PrecisionK (100)	Run time
ALS	15	150	0.01	0.01877	1927.9
Light FM	25	100	0.005	0.19489	970.15

7. Extension 2: Qualitative Error Analysis

We performed the qualitative error analysis to visualize the potential patterns and investigate the imperfect predictions. In order to visualize the learned item representation for the optimal models with rank = 250, maximum iterations = 25, regularization parameter = 0.05, we applied PCA and T-SNE on the item latent factors to reduce dimensions.

7.1. Word cloud on genre analysis:

In order to test the trend of genres of the users who produce the lowest-scoring predictions, we filtered the users whose absolute difference between the prediction and the actual rating is higher than 2.5. A word cloud is generated by counting the frequency of movie genres.

Figure 10 displays how frequently each genre appears in the lowest-scoring group. “Comedy”, “Drama”, and “Action” are the three main genres that appear most frequently.

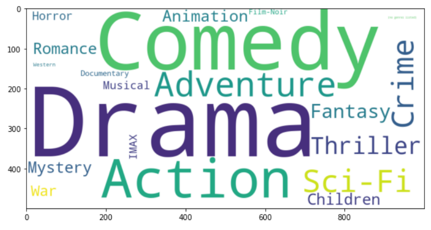

Figure 10: Word cloud for low-scoring prediction,

However, we further investigated the popularity of genres for high-scoring predictions, where the absolute difference between the predicted rating and the actual rating is smaller than 1.5, the word count for this group is displayed below:

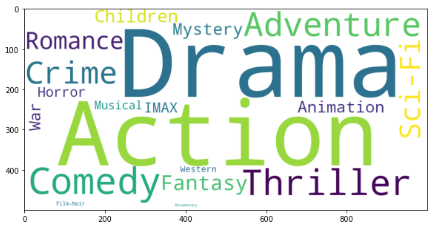

Figure 11: Word cloud for high-scoring prediction

It is clear that the trend is similar for the high-scoring group and the low-scoring group. Therefore, we may conclude that genre does not have a significant trend or an effect on the prediction model.

7.2. Rating VS Movie Latent Factors

The item latent factors were extracted from the model with movie ID and 250 latent features. Then we calculated the round average ratings for each movie. In order to visualize the learned item representation, we implemented two-dimensionality reduction methods - Principal Component Analysis (PCA) and t-Distribution Stochastic Neighbor Embedding (t-SNE). Initially, we applied PCA on item latent matrix and print the top 10 principal components with highest explained variation as below:

[0.11685, 0.06784, 0.04574, 0.04137, 0.03938, 0.03525, 0.02926, 0.02340, 0.02126, 0.02025]

The first and second principal components contain 11.69% and 6.78% information about the movie. In figure 12, it plots the x axis along with the first principal component and the y axis along with the second principal component. The color from red to purple represents the rating from 0.5 to 5. It is easy to observe a pattern along with the first PC and the second PC. We also obtained a similar pattern by t-SNE methods, as shown in Figure 13.

If under a perfect situation, we would like to see dots clustered by color. Those dots that are not positioned with the same color cluster are imperfect predictions we aim to investigate. In Figure 12, which is the PCA graph, the red box circles out some purple dots located in the green dot cluster, similarly, in the t-SNE graph (Figure 13), the red box circles out some green dots located in the purple dot cluster. For those movies, we made errors on the rating predictions. Those two graphs would help us investigate imperfect predictions for further improvement.

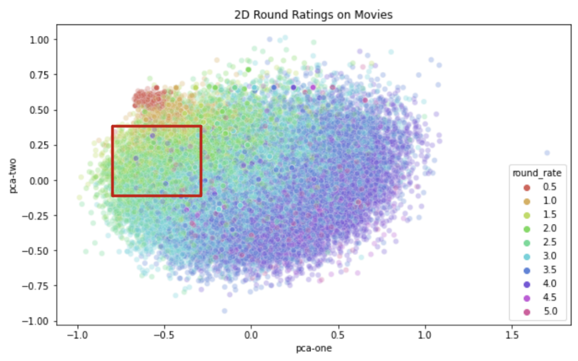

Figure 12: PCA 2D Round Ratings on Movies

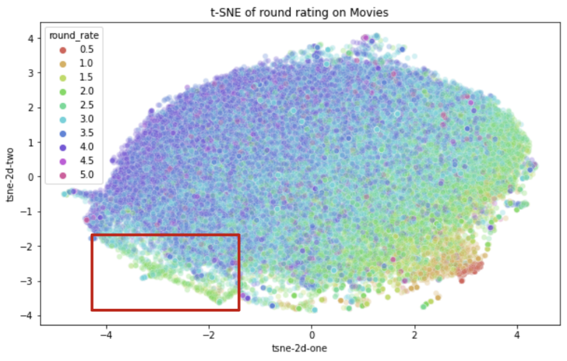

Figure 13: t-SNE 2D Round Ratings on Movies

8. Limitation & Discussion

One noticeable limitation is that the Mean Average Precision (MAP) for the baseline popularity model (0.01185) is higher than that for the ALS model (0.00863). This indicates a better performance of the baseline model. The reason for ALS not performing well could be from the data partition procedure, where our method of merging the even row number of interactions from validation and test sets into training sets would lead to a potential problem of cold start prediction. For example, there may be a lack of information for a movie rating or a prediction of a new user who does not have rating interaction in the training set. Such potential problems would be significant in decreasing the mean average precision score, which explains the performance of the ALS model was worse than the baseline popularity model.

For improvements we would recommend a model which is designed specifically for cold start situations, such a model would result in a much better performance. Or we could find a better way to partition the data to guarantee that all users, and movies are covered in the partitioned training dataset.

As for the red box we mentioned previously in Figures 12 &13, we would further investigate specifically those interactions that perform imperfect predictions. For example, we could further analyze the genres or tags of those movies and explore potentially hidden patterns in those clusters.

Furthermore, if time permits, we could deploy a hybrid recommendation model utilizing lightFM by incorporating rating and other features like tags, genres, and tag_score.

Reference:

Seeda, P. (2021). A complete guide to recommender systems - tutorial with sklearn, surprise, keras, recommenders. *Towards Data Science.* Retrieved from https://towardsdatascience.com/a-complete-guide-to-recommender-system-tutorial-wiTh-sklearn-surprise-keras-recommender-5E52e8ceace1

Link to Project Github Repository:
https://github.com/nyu-big-data/final-project-group_18