Recommender System

Yuhe Tan
Data Science
New York University
New York City
yt2336@nyu.edu

Qi Dong
Data Science
New York University
New York City
qd2046@nyu.edu

ABSTRACT

In this project, we built several recommender systems using the MovieLens Dataset, including popularity model, Spark ALS, and LightFM. We compared and contrasted different models by evaluating their ranking performance, and therefore we could recommend different models in different cases. Popularity model is recommended when people want to have a simple recommender system with no personalization. Spark ALS is suggested when people want to use Spark to build and construct their dataset. LightFM is recommended when people want to have a higher overall accuracy and spend less time on computation and fitting.

GitHub:
https://github.com/nyu-big-data/final-project-group_35

KEYWORDS

Big Data, Recommender System, Popularity Mode, Latent Factor Model, Single-Machine Implementation, Recommender Evaluation

1 Data Partition

In machine learning, we usually partition data into train, validation, and test set to perform cross validation. In the usual case, we just split the data using a ratio of 6:2:2. However, in the case of recommender system, this is not a good way. A recommender model is not able to recommend items to a user if it has not seen this user before in the training set. Therefore, we need to make sure that users in the validation set and test set also exist in the train set. Besides, we also need to make sure that no users exist in both validation and test set because the validation set and test set need to be entirely separate from each other and the validation set should be generalizing across all the users.

In order to achieve those objectives, we first separated the users into two groups based on their user ids, one for train-test split, and one for train-validation split. After that, for each group of users, we separated them into train and validation set or train and test set in a ratio of 6:4. At last, we combined the two train sets and exported them into both parquet and csv forms. The relevant code is in the file “train_test_val_split.py”.

2 Baseline Popularity Model

Recommender system is an algorithm that predicts the score or rank that a user would give to an item. Popularity model is a simple case of it, which computes the average utility of each item. Because this model predicts the same ranking for all users, there is no personalization.

In this model, we first calculated the average rating of each movie in the training set using group by. After that, we predicted a rating for every interaction in the test set based on the average ratings we have calculated in the train set and used it to calculate RMSE. In the end, we ranked the average ratings and output the top 100 recommended movies and used it to calculate the ranking metrics. The relevant code could be found in the file “popularity_model.py” in the popularity _model folder.

More specific details of the ranking evaluations are included in the Ranking Evaluations section.

3 Latent Factor Model

3.1 General Procedure

The more sophisticated recommendation model uses Spark's alternating least squares (ALS) method to learn latent factor representations for users and items. This model has some hyper-parameters that we tuned to optimize performance on the validation set, which are rank (the number of latent factors) and regParam (the regularization parameter). We also set the maximum number of iterations to 20 for better convergence.

We started from a broad range of hyper-parameters, and then zoomed in on rank or regParam until no observable differences in the evaluation score. We chose Mean Average Precision (MAP) for implicit feedback as evaluation criteria for hyper-parameter tuning, as this is the primary evaluation criteria we cared about. The details of evaluation criteria would be discussed in the Ranking Evaluations section.

Once we were able to find the best parameters, we trained the ALS model on training data and returned predictions of the top 100 movies for each user. Then we calculate the ranking metrics of predictions. The code implementations on small and full dataset can be located in latent_factor_model_small.py and latent_factor_model_largel.py in the latent_factor_model folder.

3.2 Choice of Parameters

3.2.1 Small Dataset. We first started on a broad range of parameters, regParam = [0.01,0.02,0.05,0.1] and rank =[20,50,100] and returned the MAP as implicit feedback. We plotted the result as shown in Figure 1 to get a general sense of how the change of regParam and rank would affect MAP. In general, larger rank and smaller regParam would result in larger MAP. This aligned with our understanding of the model since a large rank means more latent factors, thus the model is more complex and more flexible in prediction. Smaller regParam means less penalty to the model fitting, which introduces less bias and more variance to the model. From the graph, an increase on rank from 20 to 50 brings a significant improvement on MAP, but an increase on rank from 50 to 100 does not. Since a larger rank makes the model harder to train, we thought a rank of 50 was good enough for this model. For rank of 50, the best regParam is 0.02. To further validate our choice of parameters, we further increased rank to [100,200,300,400,500] and zoomed in the range of regParam to [0.005,0.01,0.015,0.02]. It showed that there was no significant difference on the MAP and the detailed result was recorded in the Remark file inside the latent_factor_model folder. Thus, we confirmed our choice of rank 50 and regParam 0.02.

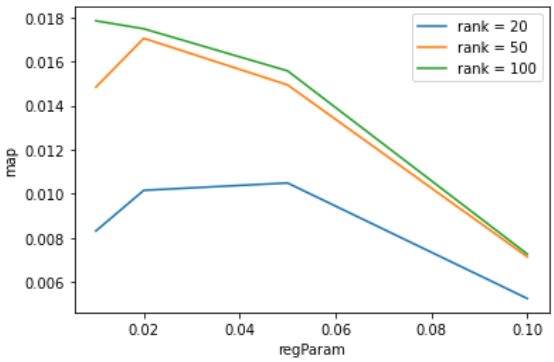

Figure 1: **Parameters Search on Small Dataset**

3.2.2 Large Dataset. Since we chose rank to be 50 for the small dataset, we would like to start our search on rank for the large dataset to be at least 50, since we expected the model would be more complex. We would also like to try a smaller regParam, since we have such a large amount of data that concern for overfitting is not as much as that in the small dataset. After taking into consideration the huge running time on the Peel cluster, we chose the range of parameters to be rank=100 and regParam=[0.05, 0.01, 0.02,0.05] and the results are shown in Table 1. The parameters with best performance are rank=100 and regParam=0.02.

MAP	regParam=0.005	regParam=0.01	regParam=0.02	regParam=0.05
rank=100	0.07163	0.09321	0.10009	0.06061

Table 1: **Parameters Search on Large Dataset**

4 Ranking Evaluations

4.1 Choice of Evaluation Criteria

For both popularity baseline and latent factor model, we would like to evaluate accuracy on the validation and test data for the predictions of the top 100 movies for each user. There are several ranking metrics to choose from. Specifically, we chose Precision at k, Mean Average Precision, and Normalized Discounted Cumulative Gain. We define each ranking metric under two circumstances. One treats the entries in the user-item matrix as implicit and compares a set of recommended movies to a ground truth set of relevant movies. The other treats the entries in the user-item matrix as explicit that incorporate numerical ratings.

The Precision at k (p(k)) is a measure of how many of the first k recommended documents are in the set of true relevant documents averaged across all users [3]. For the implicit feedback, it means that for the movies we recommend for a specific user, how many did they watch (in the test data). For the explicit feedback, it means that for the movies we recommended to a specific user and the user watched them, how many did the user rate positively. Here we treat ratings greater than 2.5 as positive. In this metric, the order of the recommendations is not taken into account.

The Mean Average Precision (MAP) is a measure of how many of the recommended documents are in the set of true relevant documents, where the order of the recommendations is taken into account [3]. Its difference in implicit feedback and explicit feedback is similar to that of Precision at k.

The Normalized Discounted Cumulative Gain (NDCG) is similar to MAP in that it penalizes highly relevant documents appearing lower in a search result, but its difference to MAP is that the relevance is reduced logarithmically proportional to the position of the ranked list. Its difference in implicit feedback and explicit feedback is similar as above.

Among these ranking metrics, we would favor MAP and NDCG over Precision at k because both of them take the order of recommendations into account. MAP is a more direct measure than NDCG in terms of introducing penalties on the order of recommendations. As a result, we used MAP as the evaluation criteria for parameter search but reported all merics when evaluating models.

We reported ranking metrics in both implicit and explicit feedback because the real-world use cases of our models would be different. In implicit feedback, we care about the views or clicks of the movies. In explicit feedback, we care about whether the ratings given by users for a particular movie is positive or not. Metrics in both feedback gave us a more thorough understanding of model performances.

We would not consider using RMSE as evaluation criteria here because ranking is not a continuous output variable, thus this is not a regression problem.

4.2 Evaluation Results

We used the above metrics to evaluate the baseline popularity model and ALS model on the small and large test data. The results are shown in Table 2 to Table 5.

	p(k)	MAP	NDCG
implicit	0.00062	0.00965	0.00057
explicit	0.40582	0.85957	0.92316

Table 2: **Evaluation on Baseline Model - Small Dataset**

	p(k)	MAP	NDCG
implicit	0.00084	0.00061	0.00116
explicit	0.25549	0.85095	0.90641

Table 3: **Evaluation on Baseline Model - Large Dataset**

	p(k)	MAP	NDCG
implicit	0.05095	0.01828	0.10303
explicit	0.05422	0.97267	0.98521

Table 4: **Evaluation on ALS Model - Small Dataset**

	p(k)	MAP	NDCG
implicit	0.03233	0.09519	0.19587
explicit	0.03665	0.99413	0.99576

Table 5: **Evaluation on ALS Model - Large Dataset**

The prediction results on the baseline popularity model is not ideal. All the ranking metrics in implicit feedback have a low value. The MAP becomes lower when we generalize from the small dataset to the large dataset. Among the movies the model recommends to each user and that user watched them, about 40 percent are rated positive, and the percentage drops to 20 in the large dataset. In general, there are no significant differences on the quality of recommendations on small and large datasets.

All the ranking metrics in implicit feedback have a larger value in ALS than those in the baseline model. This means that we achieved better prediction results. This is also expected since the baseline model produces the same recommendations to all users and is not personalized, whereas the ALS model uses latent factors to capture the interaction between users and movies. The MAP increases significantly from small to large dataset. This result could be interpreted as the large dataset enables the model to have more history data to learn from, thus having a better fit. Also, in the large test data each user has more records of

watched movies and interacts more with the training data, thus the user behavior in the test data could be better captured. The MAP in explicit feedback on the ALS model is extremely high for both small and large dataset, meaning that the model did a good job in putting highly rated movies on the top of recommendation list. But in general the model did not do well in recommending positively rated movies, as the Precision at 100 is still low.

5 Extension 1: Single-Machine Implementation

LightFM is a single-machine implementation of the recommender system. Therefore, it has its own python library and does not require the use of Spark. The input format of LightFM is different from that of ALS. In LightFM, we need to provide a utility matrix, or an interactions matrix, where each row represents an user and each column represents an item. Every entry represents the interaction weight, which is the rating in this case. Besides, LightFM requires the dimensions of the train interactions matrix and test interactions matrix to be the same in order to use the predict and predict_rank function, which means there should not be any movie id that exists in the test set but does not exist in the train set, and vice versa.

In order to construct the proper dataset to input into LightFM, we first modified the original dataset to make sure that the train_val and val set, and the train_test and test set have the same number of users and movies, and there are not other items that are not in their joint set. The relevant code is in the “modify_data.ipynb” file in the Lightfm folder.

After that, we utilized the lightfm.data.dataset object built in LightFM to construct the interactions matrix. We first fitted the user and movie ids into the dataset to create a mapping between the ids and the internal indices used by the LightFM model. Afterwards, we built the interactions and weights matrix with a bag of tuples of the form (user id, movie id, rating). The difference between the interactions and weights matrix is that every non-zero interaction in the interactions matrix has an entry of one, whereas every non-zero interaction in the weights matrix has an value equal to the rating.

After that, we performed parameter tuning and we chose a loss function of WARP because it is useful when we want to optimize the precision, according to the documentation[4]. We found that the model using the small dataset performs best when no_components (rank) is equal to 10 and when item_alpha and user_alpha (regParam) are equal to 0.02, and the model using the large dataset performs best when no_components (rank) is equal to 50 and when item_alpha and user_alpha (regParam) are equal to 0.05. We chose precision at 100 as our metric for optimization because it is a built-in function in LightFM and it is easier to calculate. We could see that the small one has a precision at 100 of 0.093 and the large one has a precision at 100 of 0.067. The relevant code could be found in the “Lightfm_parameter_tuning.ipynb" file.

We used the test set to evaluate the accuracy of the model. Precision at k is a built-in function of LightFM and therefore does not require additional steps. However, Mean Average Precision (MAP) is not provided by LightFM and we need to calculate it using other methods. At first, we tried to use the predict_rank function, which outputs a rank matrix where the [i, j] entry of the matrix contains the predicted rank of the jth movie for the ith user, and use it to output the top100 recommended movies for every user. However, manually writing code to rank the movie ids from 1 to 100 for every single user using the predicted rank matrix kills the kernel when running in the large dataset. Therefore, we manually wrote a code to calculate MAP.

First, we input the test interactions matrix into the predict_rank function. Test interactions matrix has an entry of one for all the existing interactions in the test set. The predict_rank function calculates the rank of all non-zero entries in the test interactions matrix. For example, if user 1 has an interaction with movie 2 in the test set, and the rank of movie 2 for user 1 is 5, the [1, 2] entry of the predicted rank matrix will be 5. On the other hand, if user 1 doesn’t have an interaction with movie 3, even if the predicted rank of movie 3 is higher, such as 4, the [1, 3] entry of the predicted rank matrix will still be zero. This makes sense in the context of calculating MAP, because those interactions that don’t exist in the test set, those negative examples, have a relevance score of zero when calculating average precision. We could utilize this feature of the predict_rank function to calculate MAP. For every positive example, we calculated the count of positive examples beforehand, and divided by its rank. We stopped at rank of 100 because we want to match up with the ALS case. We added the precision together, and divided by the total number of positive examples. The relevant code is in the file “Lightfm_accuracy_efficienct.ipynb”.

The comparison of accuracy and efficiency of the LightFM and ALS model is as follows when running on the modified dataset.

	Small Dataset	Large Dataset
LightFM	300 ms	7 minutes

ALS	12 seconds	14 minutes

Table 6: **Model Fitting Time on Both Models**

	p(k)	MAP
LightFM	0.109	0.182
ALS	0.051	0.020

Table 6: **Accuracy on Both Models - Small Dataset**

	p(k)	MAP
LightFM	0.068	0.112
ALS	0.034	0.098

Table 6: **Accuracy on Both Models - Large Dataset**

Because the way that the LightFM model calculates precision at k and MAP is the same as the implicit version of the ranking metrics above, we only used the implicit metrics to compare the two models. We could see that in general the LightFM model has better accuracy and efficiency than the ALS model. A possible reason could be that LightFM has a better algorithm to optimize the latent factor model compared to ALS.

ACKNOWLEDGMENTS

We thank professor Brian McFee, teaching assistants, and graders for your support this semester.
The contributions to the project are as follows. Qi is responsible for Data Partition, Baseline Popularity Model, and Extension 1. Yuhe is responsible for Latent Factor Model and Ranking Evaluations.

REFERENCES

[1] *Spark*. Overview - Spark 3.0.1 Documentation. (n.d.). Retrieved May 17, 2022, from https://spark.apache.org/docs/3.0.1/
[2] Dask. (n.d.). Retrieved May 17, 2022, from https://docs.dask.org/en/stable/
[3] Evaluation Metrics - RDD-based API - Spark 3.0.1 Documentation. (n.d.). Retrieved May 17, 2022, from https://spark.apache.org/docs/3.0.1/
[4] *Welcome to LIGHTFM's documentation!*. Welcome to LightFM's documentation! - LightFM 1.16 documentation. (n.d.). Retrieved May 17, 2022, from https://making.lyst.com/lightfm/docs/home.html