DS-GA 1004 Final Project Report: Recommender System

Lizhong Wang *lw2350@nyu.edu*
Yanqi Xu *yx2105@nyu.edu*
Fanghao Zhong *fz477@nyu.edu*

May 11, 2020

1 Introduction

In this project, we built a recommender system based on the Goodreads dataset by building a latent factor model with alternating least square algorithm. The two extensions we implemented are

- cold start with content-based models
- fast search with binary spatial tree data structure

Some basic information of the original dataset is listed in the Table 1 below.

2 Baseline ALS Model

2.1 Data Processing

Before splitting the data into train, validation and test sets, we first removed all the interactions with $rating = 0$. Since valid ratings range from 1 to 5, the interactions with $rating = 0$ indicates no interaction between this user and item. We then disregarded the users and all their interactions with fewer than 10 interactions since their interactions are not sufficient for training and evaluation.

File name	# of Records
goodreads_interactions.csv	228648343
user_id_map.csv	876146
book_id_map.csv	2360651

Table 1: Basic Information of Original Dataset

File name	# of Records
goodreads_interactions.csv	104013983
user_id_map.csv	694495
book_id_map.csv	2319634

Table 2: Dataset after preprocessing

2.2 Train-Val-Test Split

First, we split the users into 60% of train_user, 20% of val_user, and 20% of test_user using randomSplit() fuction.

Then for the validation users and test users, we divided their interactions into two halves: one half is the observed portion, which will be merged into the training data, and the other half is the unobserved portion, which is used for evaluation. We achieved this by creating row numbers after group the records by user_id, and take the interactions with odd row numbers as the observed portion while take the records with even row numbers as the unobserved portion.

Although this "row-number" method does not randomly split the data of a user into halves, we think it works here because the original data is already randomly ordered. If the original data was not randomly ordered, we would first sort the whole dataset according to a column of random numbers before using the "row-number" method. This would make sure the interactions of a user are randomly split into the observed and unobserved portions.

All files are saved into parquet files. Throughout the project, we also downsampled the entire dataset by taking $1\%, 5\%, 10\%, 25\%, 50\%$ of the users and their interactions. The summary of our train,validation and tests sets of the entire data set and 10% dataset are both listed in the Table 3 below. The mean, max and min in the table indicate the mean, max and min of the number of records per user respectively.

File name	# of Records	# Books	# Users	mean	min	max
100train.parquet	83388685	2199207	694495	120	6	38884
100validation.parquet	10309282	1056285	138962	74	5	3997
100test.parquet	10316016	1049868	138556	74	5	5797
10train.parquet	8359588	947798	69636	120	6	9508
10val.parquet	1048333	288210	14026	74	5	2407
10test.parquet	1065735	295496	13818	77	5	3169

Table 3: Train,Validation,Test of 10% and 100% dataset Statistics

2.3 Evaluation Results

Models are built and fitted using ALS from pyspark ml library. Top 500 recommendations among all items are made for each user using recommendForUserSubset() function to compare with the ground truth label. The ground truth label are the set of books that are rated by each user in the validation set. Table 3 shows that the average number of records per user in the validation set is only 74, while the recommendations are made among all training items. Hence, it is reasonable to see that precision at 500, MAP and NDCG are pretty low in general.

Two hyperparameters are tuned on the model: rank, which is the number of the user/item latent factors and regularization parameter (reg). We tuned the hyperparameters first on 10% of data to give some general direction to tuning on 100% of data. We found that a regularization parameter of 0.01 and a rank of 400 gives the highest precision at 500 on 10% of data. We also discovered that higher rank usually leads to better performance. However, the training time and memory required increases as the rank increases, which limited the possible higher rank value. The combination of $rank = 200$ and $reg = 0.01$ gives the best performance on all three evaluation metrics at 100% of data. We will predict that higher rank might give rise to even better performance. However, training the model with $rank = 300$ took longer than twice the time to train the model with $rank = 200$ and the job was constantly killed on dumbo. Hence, we settled with $rank = 200$ and $reg = 0.01$.

Rank	Metrics	reg=1	reg=0.1	**reg=0.01**	reg=0.001
10	Pre@500	6.84e-06	3.68e-05	8.93e-05	8.98e-05
20	Pre@500	6.84e-06	9.58e-05	0.00036	2.65e-04
40	Pre@500	6.84e-06	0.00027	0.00165	0.00082
100	Pre@500	-	-	0.006069	-
200	Pre@500	-	-	0.007498	-
400	Pre@500	-	-	**0.007582**	-

Table 4: Evaluation Results on 10% of data

Rank	Metrics	reg=0.1	reg=0.01	reg=0.001
200	Pre@500	4.624e-05	**0.004115**	0.002661
200	MAP	1.805e-05	**0.003328**	0.001452
200	NDCG	0.000329	**0.030325**	0.016653

Table 5: Evaluation Results on 100% of data

3 Extension 1: Cold Start

3.1 Preprocessing

We extract additional information from supplement book data (https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/books?authuser=0) about books' popular shelves and similar books. For the popular shelves categories, we pre-process them in three steps. First, we filter out the meaningless words such as 'and', 'or', 'to', etc. Second, we stem the word to their base or root form. Third, we group similar popular shelves categories together based on Jaccard similarity (Jaccard>0.7). For example, categories 'interest book' and 'an interesting one' will be mapped to the same group. After the pre-processing steps, we reduce the book shelves categories from total 1591777 to 635684. In the combined shelves' categories, there are some rare categories which only applied to a few books, so we will filter them out because the existance of these rare categories will not affect the cold start results but slow down the training time. We set the threshold to be 1000, which means all popular shelves categories we included are at least shown on 1000 different books among 2.3M books. After thresholding, we finalize 9372 popular shelves categories. When we pre-process popular shelves, we use additional packages 'nltk', 'textblob', and 'graphframes'.

3.2 Training

In our baseline models, we apply ALS model to calculate user matrix and item matrix with 20 latent factors and 0.01 regularization, but PySpark does not have module to train multivariate linear regression yet, so we have to train univariate linear regression separately for each of the latent factors. To improve time and resource efficiency, we choose to apply ALS model trained with 10% data with 10 latent factors and 0.01 regularization to do cold start extension. We set latent factors as model's dependent variables and popular shelves as models' independent variables, and train linear regression models and lasso regression models separately. In lasso regression models, we choose regularization to be 0.01. If we have more time, the regularization term could be tuned to improve the performance.

From the regression models, we can calculate latent factors for new books based on their associated popular shelves. However, some books do not have associated popular shelves, so we also utilize similar books in the supplemental data to calculate books' latent factors. First, we find the book's similar books and extract latent factors for the similar books. Then, we calculate average latent factors for the similar books and then assign the average latent factors of these similar books as the new latent factor for the new book. If the new book has both popular shelves and similar books in the supplemental data, then we will calculate the weighted latent factors for the new book based 60% on popular shelves and 40% on similar books.

3.3 Results

We utilize 70% books to train linear and lasso regression models and hold out 30% books as new books we never seen. We utilize methods above to calculate latent factors for new books, and then recommend books to users based on the new latent factors. The table 6 summarizes the performance.

	Pre@500	**MAP**	**NDCG**
Baseline	8.93e-05	9.33e-06	0.000299
Linear Regression	9.30e-05	5.54e-06	0.000305
Lasso Regression	9.37e-05	7.91e-06	0.000314

Table 6: Performance Comparison among Original, Linear Regression, and Lasso Regression

After utilizing more information to do cold-start recommendation, we notice that the performance slightly improves. It might be the effect of linear/lasso regression and similar books cancelling out noises in latent factors of original training books.

4 Extension 2: Fast Search

We used Annoy package for this extension. Annoy achieved fast search by building up a binary spatial tree using random projections method that we covered in Lecture 10.

4.1 Build Annoy Search Tree

Before we build Annoy search tree, we need to save the ALS model so we can retrieve the item factors. Then we could build an Annoy search tree with Dot Product distance using all item factors of the ALS model.

4.2 Query with Annoy Search Tree

First we retrieve the user factor for each validation user. Then we use Annoy search tree to find the nearest 1000 item factors to this user factor, and these factors form the candidate set of this user. Afterwards, we calculate the predicted rating scores for each candidate book item. Finally, we sort the candidate books by the predicted rating scores and recommend the top 500 books.

4.3 Performance

We tested the performance of fast search implementation using the history genre data on our local machine. We used different fraction of items to compare the running time and ranking metrics of Baseline model and Annoy search tree. We excepted the difference of the querying time between these two methods will increase as the number of items increase because while the Baseline model has to compute a predicted score for each item, the fast search implementation only compute for the candidate set. The result is summarised in Table 7. The full result is in the Fast_search-compare_result.txt file on GitHub.

	#Books = 27944	**#Books = 84096**	**#Books = 140037**	**#Books = 210504**
Baseline	PK = 1.6e-5, T = 3.8min	PK = 6.6e-6, T = 14.9min	PK = 5.2e-6, T = 36.7min	PK = 4.2e-6, T = 69.1min
Annoy	PK = 2.2e-5, T = 2.5min	PK = 1.5e-5, T = 5.1min	PK = 1.6e-5, T = 6.7min	PK = 1.3e-5, T = 8.5min

Table 7: Fast Search Performance

The result confirmed our initial expectation of the running time. As the number of items increases, the query time of the Baseline model increased significantly. In contrast, the query time using Annoy search tree only increased moderately. However, we were surprised that PK of fast search is higher than PK of Baseline. We suppose this is because recommendForUserSubset() can recommend books that the user has already rated, which the Fast-Search implementation does not.

5 Conclusions

The final model fitted with $rank = 200$ and $reg = 0.01$ is evaluated on the test set and the results are shown in Table 8. Ideally, we will need to combine the train and validation data to fit a new model with tuned hyperparameters. Due to limited time and resources, we haven't done this part successfully. This is something we should do in the future.

Pre@500	**MAP**	**NDCG**
0.004126	0.003241	0.030107

Table 8: Evaluation on the test set

Lizhong Wang: Data process/split/baseline improvement, fast search, report write-up
Yanqi Xu: Data process/split, baseline, report write-up
Fanghao Zhong: Hyperparameter tuning, cold start, report write-up