Implementation of A Recommendation System (ALS Approach)

May 12, 2020

0.0.1 Student Name: Shensheng Chen (solo project)

0.0.2 SID:15781358

0.0.3 github: https://github.com/nyu-big-data/final-project-solo-rec-system

0.1 Introduction

Recommendation system plays an important role in our daily life and it is almost everywhere. When you watch Netflix, Netflix recommend films/shows based on your watch history. When you play tic-tok, the algorithm will recommend you short videos based on your 'like it' history. When you shop on Amazon, you will be much more likely to click on 'people bought this items also bought' recommended items. In our project, we have data from goodreads, a famous book-rating website, and our goal is to develop and evaluate a good model that could recommend books to the reader based on his/her ratings of previous books he/she read. We will first develop a model that works on hadoop/spark environment and can do parallel computing and then we compare this model to single machine model developed by library like lightFM.

0.2 Split, Subsample and Data Preprocessing (See setup.py for all data preprocessing codes)

We open the goodreads_interactions.csv file and we see that the file contains tuples of user-book interactions. For example, the first five lines have columns user_id, book_id, is_read, rating, is_reviewed. Since we know that rating is the most important feature we need to simplify the problem we will only focus user_id, book_id, rating in our basic model. Therefore we delete is_read and is_reviewed columns.

We now need to do two actions to deal with our hundreds of millions interactions data. First, the interactions data is so large with over 800k users it's not wise to directly train our model based on such large data thus we need to subsample our data. We filter out users that read less than 10 books because they are not representative and have no useful information. We can simply use limit function to sample the interactions. Once we get our sample, we need to consider how to split our sample. We can split the sample into three sets, training set, validation set and testing set. When we do spliting, it is clear that we cannot just split the interactions because otherwise it will raise a very awkward senario: we train the model and the training set doesn't contain certain users, but we need to give recommendations to the users even though we know nothing about those users. Thus the best way is to split the interactions by users into 60/20/20. Even with the validation set and testing set, when we train the model we will cut the validation set into two parts, the odd index part and the even index part. Odd part we merge them into the training set so that the model will at least know some facts about the users in the validtion set thus can give better recommendations

than know nothing at all. Split the interactions based on index odd/even is not trivial. We brute-force maunally add a new column id_x to the original df by using rdd.zipwithindex function. Then we filter df with id_x %2 ==0 and df with id_x %2 ==1. By doing so we finish our implementation of spliting the interactions into training,validation,and testing. Although the original df was sorted by user_id, to make it consistent, we decide to also sample the interactions by users.

In summary, given an integer count $<$ 800k represents the count(users), our setup.py can generate a sample df with count users and save it as samplecount.parquet for further use. also, we have validtrainingcount.parquet that merges training set and odd half of validation set. validvalidcount.parquet that represents even half of validation set. The same thing applies to testing sets. In summary, the setup.py codes implement the sample and spliting of the original interactions csv, save them as parquet for model development and evaluation use. In my NYU HPC cluster, I have count = 1000,10000,100000, and 500000 users. In my setup.py file, I implement split_interactions function, dividesample function, and writeparquet function. Those functions are very important to sample and split our origianl data. in pyspark, once you import setup, use setup.start(count), it will automatically sample the data based on count users and do split for you and save the files to the hdfs system. Once it finishes its job, it will show "we are done!"

0.3 ALS Explaination, EDA, Latent Factor And Rank Guess (findbookdistancematrix.py)

Before we start to develop our model. We need to first understand how our model works and the model uses what kind of algorithm. We already have our simplied interactions data frame with three columns: user_id, book_id, rating. In other words, we have a sparse matrix M such that the rows of M are user_ids, and the columns of M are book_ids, and the entrees of M are the rating. For example, if we want to know the user i's rating on book j, we simply get the entree of $M_{i,j}$. Therefore, our goal is to find $\hat{M}$ based on M' where M' contains only part of the info of M, what's more we need $\hat{M}$ to fit M as much as possible.

The ALS method is a very compute efficient way to compute $\hat{M}$ and make it fit M. Basically the algorithm states $\hat{M} = UV$ where U is the user matrix with user rows and d columns named 'latent factors related to users' and V is the book matrix where rows are 'latent factors related to books' and columns are books. We initialize U, V (thus fix latent factor dimension, rank d) and compute $M - \hat{M}$. Then we regard U as fixed and trying to find V that make $M - \hat{M}$ as small as possible. Once we find such V, we fix V and then search the best U that minimize $M - \hat{M}$. Repeat this process finite times we will see that the $M - \hat{M}$ error would be samll enough to reach the threshold ϵ. Then we finish our process and the $\hat{M}$ fit M very well. This ALS algorithm is essentially gradient descent (or Newton's method if we see it as solving equation in Numerial Analysis)

However, one question raises before we try to develop our model. What's the possible range of our rank d? Clearly we can make d as large as possible so that ϵ would be really small. However, large rank would takes much more time to compute and we might overfit the matrix. Thus we need to give a reasonable rank guess based on EDA of our data.

What's the meaning of the rank (the number of latent factors)? Clearly we can regard them as features of users and books. For example, in the U matrix, a specific row ($1 * d$ vector) represents a user, and that's all the info we have for the user. Similarly, in the V matrix, a specific column ($1 * d$ vector) represents a book, and that's all the info we have for the book. Thus the rank is the number of features of user and book. The feature numbers of user and book are consisent because of two reasons. First, since we need to do UV multplication we have to make the column number

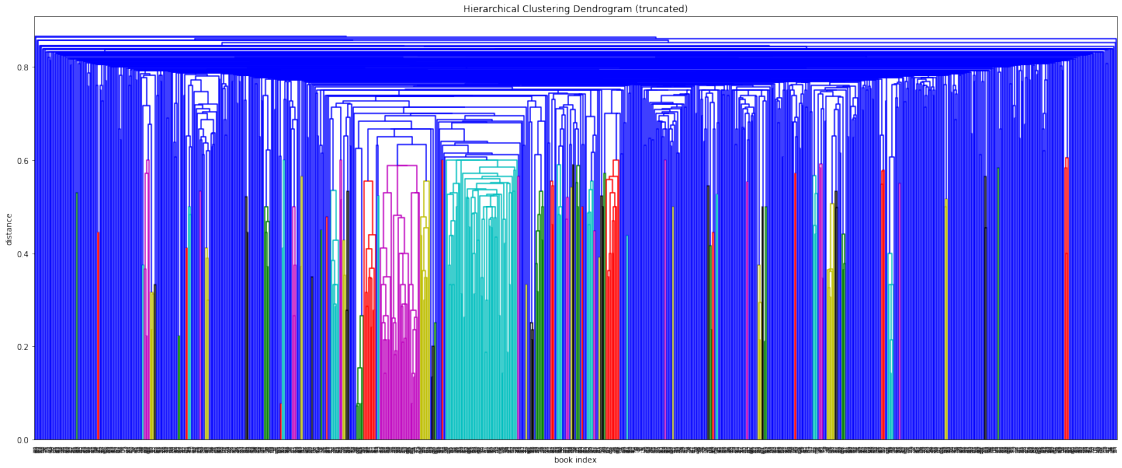
of U and the row number of V match. Second, I believe there is a duality in our decomposition. In other words, similar books are clustered by similar users and 'You are the book you read'

Before we develop the model, we can do EDA on interactions data, cluster the books based on users who read them to predetermine the rank (number of clusters). First, we still filter out books are not popular (read by less than 10 users). We also filter out ratings ≤ 3 because we only want to cluster books that users like and low rating books are not representitive. Then we use collect_list function to collect users based on books. In our sample with 800 users, we get the following collect_list(user) data frame with 966 books: we can have millions of book list but since we will compute the distance matrix and do hierarchical clustering visualization it's best to keep our bookset relative small but still representitive enough to give us a good rank guess.

To compute the distance between book_i and book_j. We define the distance as modified verison of jacarrd distance. the distance

$$Dist(b_i, b_j) = 1 - \frac{\#(U_i \cap U_j)}{\#(U_i \cup U_j)}$$

with the distance defined we induce a distance matrix. Clearly, the matrix is symmetric thus we only need to consider the upper triangular matrix. With scipy library we can easily get the matrix and use the distance matrix to do hierarchical clustering.



We can use this clustering graph to estimate our rank.First we know that, if two books are similar they must have small distance between each other by our defination of jaccard matrix. Thus we are not interested in the blue lines because clearly those are books whose distance between each other > 0.8. we can set up a cut off threshold as 0.5 and count color clusters different than blue and below 0.5, like the green color, yellow color, red color etc. I found around 30 clusters below 0.5 Jacarrd distance. Thus we finish our estimation of rank and I will make our rank guess as 30 and tune our rank hyperprameter around 30 by step 1 in our next chapter- Model Evaluation.

0.4 Hyperparameter-Tuning & Model Evaluation (evaluation.py)

With our estimate rank guess = 30 we can start to build our model and tune hyperparameters. We tune two hyperparameters: the rank and the regularization parameter lambda. My plan is to use

30 as base for the rank, and check rank from 25 to 34 with step 1. For regularization parameter lambda, I plan to use 0.01,0.1,1 as pealty coefficents. Thus we will train $10 * 3 = 30$ models and see which one performs overall best over validation and testing set.

Since the computing resource is very limited on NYU dumbo cluster and my network is not good enough for me download raw data to my personal computer, my data size is relatively small and I can't tune the parameters for a large range. For example, I spent >6 hours to tune those 30 models with 100k users (770k+ users totaly for whole data if we drop users who read less than 10 books). Thus I really want to tune rank from 25 to 34 and use the whole 770000+ users as my data but I couldn't manage to do that. (It will take days to compute on 3 cores and 6g ram dumbo cluster) The cluster constantly failed if I try to run my evaluation.py code on 770k+ users but at least for 100k it works.

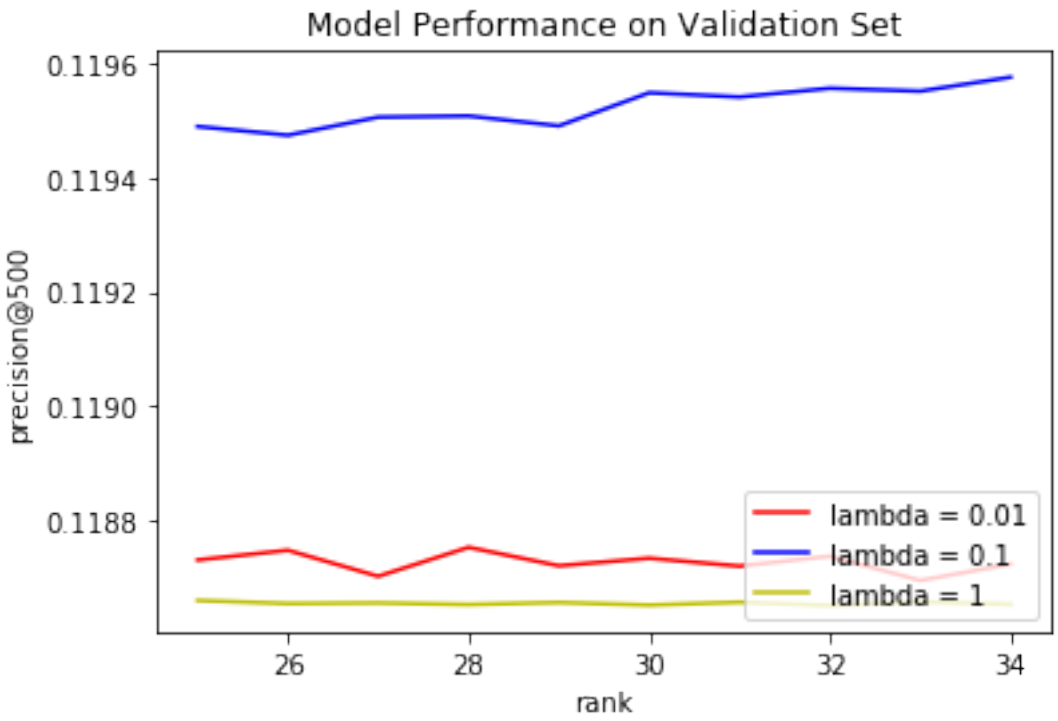
Once we have our models, we must have a 'metric' to value how it performs. We can do two kinds of metrics. One way is to do regression metric: given the model and testing data, we do predictions (use transform function) and compute the difference of the predictions and true values (RMSE-L2 sense). Another way is to do ranking metric: given the model and testing data, we pick the users from testing data and ask the model to give specific recommendations (RecommendforAllUsers function) to the users repectively, and we compare our recommendations with the true book set (top 500 sense, can do ndcg,precisio@500,etc). However, both methods have significant drawbacks. If we do regression metric to evaluate our models, then this project is no longer 'recomendation system project' but a traditonal 'regression analysis project'. If we do ranking metric to evaluate our models, since for each user they read relatively extremely small amount of books comparing with the whole book shelves, the recommendation ranking metric would be extremely low (ndcg 0.002 etc) since all the books that the system recommends the user would not read thus we have no data to compare it! That's a very serious problem.

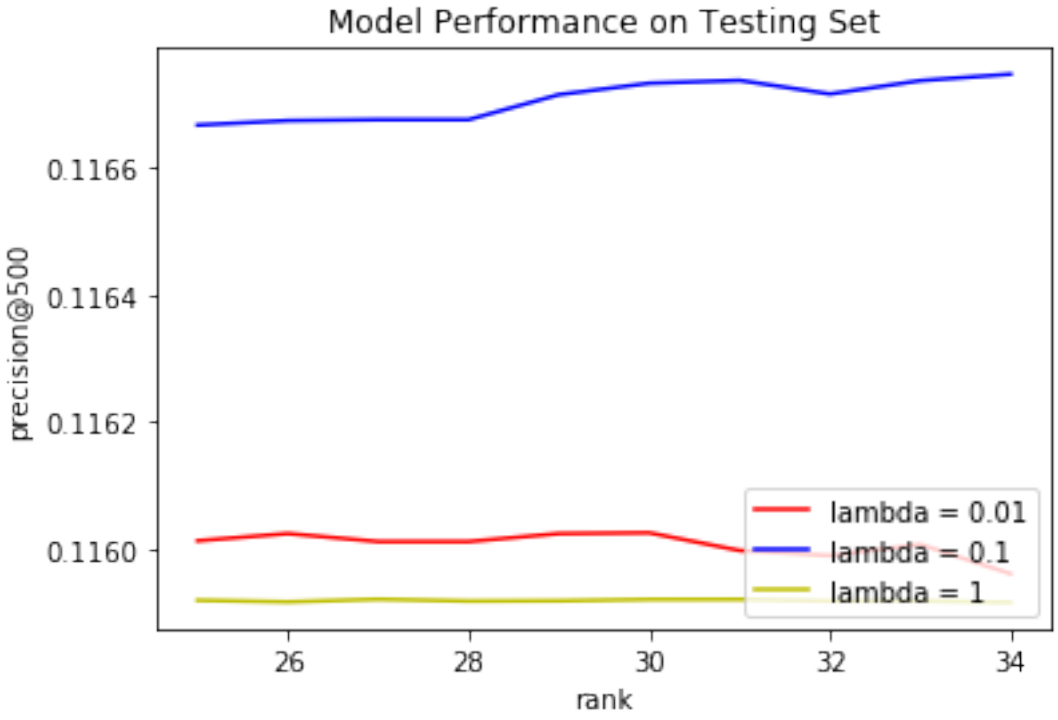
My idea to approach this problem is: I am still doing recommendations but 'restrict' to the testing/validation books. Yeah, **it's a bit like you travel from the future and claim that you have the ability to 'predict' what will happen in the future but I think I have no other better ways to approach this problem without getting insignificant statistics or deviate the problem.** So my approach is to first predict the ratings with our model on the testing/validation set. Then we do two sorts with our user_id, book_id, rating, prediction_rating DataFrame. The first time we sort out the true book set, by groupby(user_id), collect_list(book_id, rating) then sort the collect_list based on rating desc. Similarly, we get our sorted prediction set.

with our truth_set and our pred_set we can do ndcg, map, precision@500 etc ranking metrics. One thing to note is that since we 'travl from the future', our statistics would be 'too good to be true', for example, the ndcg would always be 1. My way to to cut the real book set to make its rating >= 3 and then compute ndcg, map, precision@500. To me personally a 'too good to be true' statistics is always better than pure recommend approach a insiginificant ndcg 0.002 anyway!

With the computed ndcg, precision@500, map we can evaluate our models. Here is the model performance on validation set.

[22]: `Text(0.5, 1.0, 'Model Performance on Validation Set')`


Model Performance on Validation Set
0.1196
0.1194
0.1192
0.1190
0.1188
precision@500
26
28
30
32
34
rank
lambda = 0.01
lambda = 0.1
lambda = 1


Model Performance on Testing Set
0.1166
0.1164
0.1162
0.1160
precision@500
26
28
30
32
34
rank
lambda = 0.01
lambda = 0.1
lambda = 1

Thus we see that first clearly $\lambda = 0.1\$\$$ is the best lambda parameter among those three. Second, precision@500 fluctuates when the rank changes but rank = 34 is a pretty good rank on both validation and testing set. In summary, among all those 30 models I prefer the model with rank = 34 and hyperparameter = 0.1. It is clear that we can always continue to make rank larger and larger or testing more regularization lambdas but since tuning high ranks will be significantly more time consuming I think we can happily stop here. We also observe the fact that $\lambda = 0.01$ fluctuates much more violently than $\lambda = 1$ This is because $\lambda = 0.01$ has less penalty on the complexity(overfit drawback) of the model thus it will more likely to respond to the data more violently.

0.5 Comparing with LightFM (lightfm_eval.py, evaluation_rec_version.py, lightfmvsALS.py)

LightFM is a Python implementation of a number of popular recommendation algorithms for both implicit and explicit feedback, including efficient implementation of BPR and WARP ranking losses. It's easy to use, fast (via multithreaded model estimation), and produces high quality results.

The differnce between LightFM and our ALS approach is that our ALS approached is integrated in the Spark/Hadoop environment but LightFM is more like for a single machine algorithm (though you can assign different processor cores through the num_threads parameter , for parallel computing, but it is still runing on a single machine) . In our lightFM model, we choose 'warp' loss function, no_components = 28 (the rank), and $\alpha = 0.1$ as our λ. **Since LightFM implements only 'recommendforallusers' function, our model developed in evaluation part is not consistent with LightFM thus we cannot directly compare them. I decide to write another verision of evaluation.py that uses RecomendforAllUsers function thus we can compare those two model statistics (precision@500).** Also since LightFM won't ignore ratings <= 2 we won't filter those items in our modified evaluation.py version

We observe that lightfm runs relatively fast on small dataset, even faster than ALS method when we pick user numbers as 1000 and 10000. However, when the user numbers reach 100000, the lightfm method would be extremly slow. Actually, I never get the runing time and statistics from lightfm if the data is based on 100000 users. However, for our spark ALS method we dont have to worry about that because all the data is distributed in the spark environment instead of simply put them in ram we can always get the ALS statistics. Although ALS is not that fast, we avoid the possibility of outofmemory.

[24]:

	dataset	ALStime	ALSprecision@500	lightFMtime	\
0	1000	96.11298036575317	0.013809999999999992	40.733498096466064	
1	10000	1292.130049943924	0.003711	377.86484241485596	
2	100000	large number	?	error	

	lightFMprecision@500
0	0.0002
1	9.7000004e-05
2	error

In summary, ALS has better the precisio@500 statistics than lightfm but when the dataset is small lightfm performs faster than ALS. When the data is large, ALS still works due to the unique distirbuted design of spark enviroment but lightfm start to raise error due to out-of-memory problem. Our conclusion is that for a large amount of data ALS is very good and the only choice. For a

small data, lightfm is pretty good because of its convinence and light build.