RECOMMENDER SYSTEM PROJECT

May, 2020

Frank Jiang
netid: hj1325
Email: hj1325@nyu.edu
New York University
DS-1004 Big Data

1. Basic Recommender System

1.1 Data splitting and subsampling

In this project, we will be using the Goodreads dataset collect in late 2017 from goodreads.com by Mengting Wan and Julian McAuley. This metadata contains information about '*876,145* users; *228,648,342* user-book interactions in users' shelves (include *112,131,203* reads and *104,551,549* ratings)'. We can split this metadata set into a training, validation, and testing dataset by the percentage of 60%,20%,20% accordingly. Since the recommender system cannot predict items for users with no history, we use half of the interactions from validation and testing for training while holding the other half for validation and testing purposes. We create a window partition while adding a row number to validation and testing, then split the data frame by even and odd row number. All data joining is completed by SQL in Spark. In order to increase processing speed, we saved all CSV files to parquet in the Hadoop cluster for later use. Training, validation, and testing are saved as 'train_df.parquet', 'validation_df.parquet', and 'test_df.parquet'. A precise data splitting process can be found in Data_Splitting.py.

1.2 Basic Implementation

The recommender model was created using the Alternative Least Squares method (We will refer this as ALS method) in Spark, which is a method for filling missing entries in collaborative filtering to learn latent factor representations for users and items. We fit the model on the train_df.parquet dataset and evaluate it on validation_df.parquet dataset based on the top 500 recommendations for each user in it while adjusting the hyperparameters such as regularization parameter(regParam), *baseline* confidence(alpha), and the number of latent factors(Rank). The model defaults to handle implicit feedback by setting implicitprefs to true. We are only able to fit the model with 60% of the training data with interactions due to the instability and limited storage in the Dumbo cluster.

1.3 Hyperparameter Tuning

We will use Final_Project_Train.py to train and tune our hyperparameter for model evaluation to select a best-fitted model. After initial experiments and play around with the value of parameters, we believe that the following values of hyperparameters are reasonable to use to tune our model.

regParam= [0.1,1,10,100] Alpha= [0.1,1] Rank= [10,100]

We initially decide to evaluate the model performance by using Precision At K and Mean Average Precision methods available in Spark.mllib.evaluation. After multiple iterations and implementations, we decide to move on to applying RegressionEvalutor for Root Mean Square Error as the new critique for model evaluation due to the long RDD run time on the Dumbo cluster. (Old evaluation methods are saved as notes in the python file.) We have excluded hyperparameter higher than 100 to save run time. We create a loop that will return the result for each try and save the best performed ALS model for future use. Overall, there should be 16 different combinations of hyperparameter values.

1.4 Evaluation

Based on our results, the best-performed model is using the following set of hyperparameter:

regParam=[0.1] Alpha=[1] Rank=[100]

These will be saved as our default model hyperparameter in later testing and further analysis. The full results are present in the following table for evaluating our model performances.

Table 1: Hyperparameter tuning results using validation data

regParam	Alpha	Rank	Root mean square error
0.1	0.1	10	2.886630293
0.1	0.1	100	2.865060762
0.1	1	10	2.818645918
0.1	**1**	**100**	**2.770724736**
1	0.1	10	2.924946435
1	0.1	100	2.924747321
1	1	10	2.862796205
1	1	100	2.846223283
10	0.1	10	2.925541289
10	0.1	100	2.84128856
10	1	10	2.832885596
10	1	100	2.801823747
100	0.1	10	2.915341223
100	0.1	100	2.925541289
100	1	10	2.892039283
100	1	100	2.872948374

After further analyze our model performance against testing data, we can conclude our model with a root mean square error of 2.7718, a precision at 500 of 0.01, and a mean average precision of 0.0093. These results are shown in Table 2 below.

Table 2: Hyperparameter tuning results using testing data

regParam	Alpha	Rank	Root mean square error	Precision	Mean Average Precision
0.1	1	100	2.771785827	0.010341	0.0093047

In the next section, we will discuss the tradeoffs and concessions of the three parameters. Our three hyperparameters might not be precise enough for tuning, so the optimum hyperparameter values may lie somewhere in between. In such a case, further analysis and detail scoping should be the next stage of recommender development.

1.5 Concession and adjustment

There are some main constraints during our model development and evaluation. One of the primary ones was the size of the training data. We are only able to fit 60% of the training data due to the calculation capability of the Dumbo cluster. We apply several strategies to downsampling the data. Except for the one mentioned before, we also select rows that contain user only exists in validation and testing dataset. We excluded ratings that are 0 since books with a rating of 0 normally are not viewed by the user yet. This could potentially impact model performance since not all data are included in the model.

Another one would be that the failure of using precision_at_k and mean average precision as the selection critique. Our initial analysis plan was to incorporate these evaluation methods. Precision is more effective and straightforward when it comes to ranking algorithm evaluation. Regression Model evaluation is used more often when it comes to predicting a continuous output variable, which is slightly different from the ALS method of our recommender system. Therefore, we could see how this could potentially affect our model selection optimization.

2. Extension: Fast Search

For the extension feature, we decide to implement the fast search feature extension. During our model training process, we noticed that it is time-consuming to recommend for all users. Therefore, implementing a tree-structure spatial data will most likely accelerate the query time. After the literature review, we decide to use the annoy library, which creates an index for approximate nearest neighbors search.

We subsampled 1% of the users from the test set and generate recommendations for each user. Then we discovered a range of hyperparameters in the annoy library and analyze its influence on the searching time. By tweaking the number of trees built during indexing(n_trees) and the number of neighbors to search(search_k), we can explore the trade-offs of these hyper-parameters. Table 3 below shows the result of permutation. Run-time for generating recommendation without using annoy library are saved as N/A.

Table 3: Hyperparameter tuning of n_trees and search_k in fast search

n_trees	search_k	time_spent	precision	mean average precision
N/A	N/A	203.43	0.022103	0.031775
10	-1	135.77	0.013045	0.023043
10	5	145.31	0.013045	0.023043
10	10	156.53	0.013045	0.023043
15	-1	133.27	0.014345	0.267384
15	5	146.38	0.014345	0.267384
15	**10**	**132.42**	**0.014345**	**0.267384**
20	-1	146.24	0.014013	0.253059
20	5	151.34	0.014013	0.253059
20	10	160.29	0.014013	0.253059

Based on our results, we can find that implementing a fast search feature significantly decreases the recommendation generation run-time. The lowest run time of recommendation is completed with n_trees=15, and search_k=10, which decreased the run time by 35%. Interestingly, in our model, n_trees seems to be the only hyperparameter that affects the precision of the original model. Also, n_trees seems to be the primary factor that affects elapsed time.

Since we only subsampled 1% of the test dataset for the fast search feature extension, it will be worth investigating the effect fitting a large sample of data in the future.

Appendix

Data Storage

All data has been preprocessed and saved in the following repository on the Dumbo cluster as well as the Model indexer and ALS model which might be used during installation.

Training data:	'hdfs:/user/hj1325/final-project-final-project/train_df.parquet'
Validation data:	'hdfs:/user/hj1325/final-project-final-project/validation_df.parquet'
Testing data	'hdfs:/user/hj1325/final-project-final-project/test_df.parquet'
Model indexer	'./home/hj1325/final-project-final-project/model_indexer'
ALS model	'hdfs:/user/hj1325/final-project-final-project/final_model

Installation instructions

***Basic Recommender System*:** The basic recommender system includes three components, Final_Project_Train.py, Final_Project_Test.py and Data_Splitting.py.

--Data Splitting: - spark-submit Data_Splitting.py path_to_data_file

This program clean, preprocess and split the original data into training, validation, and testing data with the percentage of 60%,20%,20% accordingly.

--Training and Tuning Model: - spark-submit Final_Project_Train.py path_to_train_data path_to_validation_data path_to_test_data path_to_save_model tuning option

This program takes the input of the training data, validation data, and testing data, preprocess the data to determine the necessary user row for creating the model. The default parameter 'tuning' is set to off to test through the entire pipeline that creates an index, transform, then fit the ALS model with training data. Set tuning=true to activate hyper-parameter tuning.

--Testing Model: - spark-submit Final_Project_Test.py path_to_test_data path_to_model_indexer path_to_als_model

This program takes the input of the test data, loads a model indexer(string), a fitted ALS model, and evaluate the performance of the model against test data.

***Extension: Fast Search**:* This extension requires the use of packages annoy, which is available through pip install annoy

--Evaluate Elapsed Time and Precision: - spark-submit Extension_fast_search.py Path_to_test_data path_to_model_indexer path_to_als_model limit option

This program takes the input of the testing data, a model indexer(string), a fitted ALS model to generate trained brute-force search model. The limit option determines the proportion of testing data we are using to train our fast search extension. (limit defaults to 0.01)