DS_GA1004 Big Data Final Project

Enyi Lian (el2986@nyu.edu)
Jimin Tan (jt3545@nyu.edu)
Zian Chen (zc674@nyu.edu)

2019/05/18
Instructor: Brian McFee

Contribution:

Enyi Lian: preprocessed the datasets, coded for the baseline model and evaluation, tuned hyper-parameter, worked on “log compression” extension, wrote the final report;

Jimin Tan: tuned hyperparameter, worked on “exploration” extension in visualization with t-SNE, wrote the final report;

Zian Chen: tuned hyperparameter, worked on “dropping low count value” extension and wrote final report.

I. Introduction

In this final project, our group attempted to build a collaborative filtering recommender system to provide recommendations for users. Spark applies alternating least squares (ALS) algorithm to learn latent factors representations for users and products. We implemented the ALS model embedded in pyspark.ml to assist in solving the large-scale recommendation problem with given datasets in parquet format. Our work included data preprocessing, hyper-parameters tuning, implicit feedbacks modification, and visualizing latent space distribution.

II. Basic Recommender System

The first step is to preprocess the data. Currently, the DataFrame-based API for ALS algorithm in Spark only supports integer for user and item ids. Since the original ids for user and tracks are in string format, we converted them to the numeric representations for Spark ALS model to work. However, we ran into another problem while saving the indexed data: the training dataset was too large for us to store a copy in hdfs directories, so we decided to downsample the training data.

The validation and test data were not too large so we kept them the same. The training data has a large scale and contained complete listening behavior histories of approximately 1M users and particle histories of 10K users in the validation data and 100K users in test data. Since the training data has both evaluation and testing users, the baseline model evaluation/testing would not suffer from the cold-start problem if we use the whole training set. To ensure our sampled training data do not have cold-start problem either, we explored the training data and figure out a reliable sampling method. Our sample should include all partial histories of users in the validation and test data along with additional histories of the remaining users. After left joining the training data and the other two data separately, the records of the corresponding users in the validation data are over 130K and the corresponding records of test users are around 13K. Considering the computation speed and storage limitation, we selected 500K records from the end of the table as our sample training data.

Then, we started to index user and track columns. We encoded the two string columns, “user_id” and “track_id”, to index representations with “StringIndexer”. The same fitted indexer from training data was then applied to transform all three datasets. For the convenience of frequent usage of the indexed datasets in the following assignments, we saved the indexed results at HDFS in parquet format.

After data preparation, we built the basic ALS model and tune parameters to optimize the model performance based on the ranking metrics evaluation. The three hyper-parameters we will focus on are “rank”, “regParam”, and “alpha”. We applied default settings of the remaining parameters in the ALS model, besides the three parameters of interests and “ImplicitPreference”. Setting “ImplicitPreference” as True indicates the model to treat “count” as the confidence representations in observations of users preferences, rather than to regard “count” of listening as

the direct reflection of explicit ratings given to songs. With respect to the hyper-parameter tuning part, we constructed a grid of parameters to search over: rank (dimension of latent factors) in {5,10,15}, alpha (implicit preference formulation) in {0.1, 1, 10}, and regParam (regularization parameter) in {0.01, 0.1, 1}. The chosen search ranges included all default values of the three hyper-parameters. In the next step, we manually built models over the 27 combinations of the hyper-parameters and evaluate their performance on the validation data. The high volume workload of a single model rendered us to abandon the MLib model selections tools, such as "CrossValidator" and "TrainValidationSplit". The model training with one hyper-parameter setting took about 7 hours on dumbo, thus mutually tuning was a more realistic choice in our opinion. We trained our models through the subsampled indexed training dataset and compared the ranking metrics on the complete indexed validation dataset. In this project, we picked Mean Average Precision (MAP) of top 500 items for each user as the model selection metrics and also check the optimal model on test data with the precision at 500 recommendations. Both are commonly used for evaluating recommendation system performance. The difference between the two precision calculations is whether the orders of the recommendations take into account.

The default setting of the ALS model in pyspark.ml (rank=10, regParam=0.1, alpha=1) returns a baseline result of MAP at 0.03175. In general, as rank got larger, the MAP got higher; while regParam=0.1 models typically output the best precision, when controlling for other hyper-parameters. At the same time, the effect of alpha on precision is a bit ambivalent. In some cases, the precision peaked when alpha=1 given the other hyper-parameters stay the same, but it also happened when alpha=10 gives the best precision. Overall, we discovered our baseline optimal model at alpha=10, regParam=0.1, and rank=15. The tuned model improved the MAP by 19.24% and achieve the precision of 0.03784 on the validation data. The model predictions on test data gained a precision at 500 of 0.008565 and MAP at 0.03451. The tables of all tuning evaluations could be found in the Appendix.

III. Extension

Continuing to improve the performance of the basic recommender system, our group implemented two extensions into the picked baseline model, "Alternative model formulations" and "Exploration".

A. Alternative Model Formulations: dropping and log compression

Modifying the count data was supposed to improve the behavior of implicit-feedback modeling. We first dropped low values from the "count" column. Listening to a song only once might not be a reliable indicator to identify if the user is genuinely interested in the audio track or not. Around half records in the subsample training data is with the count of 1. The connotation of "dropping" was quite vague here, so we executed it in two ways. Removing all "count" value at 1 directly from the subsampled training dataset may lead to a cold-start problem because some users' full histories might be removed. Luckily, all users are left. In the other method, we

transformed the count of 1 to 0 for keeping all users left in the training data. Unfortunately, two models built from the modified datasets turned out to underperform than the baseline model. We picked the best hyperparameter combination, fitted the ALS model to the modified training dataset and got ranking metrics decreased by 13% with respect to the model trained from the dataset without count 1, from 0.038 to 0.033. The ranking metrics MAP dropped by 26% from 0.038 to 0.028 in the case if we replace 1 with 0. Due to the poor performance of the two models extracted from the modified datasets on validation sets, we gained a lower evaluation result from the test data as expected.

Moreover, we replaced "count" as the implicit rating towards items by its log transformation. The "log" function in SQL compute the natural logarithm of the given values. The log compression does not bring any improvement in the optimal baseline model performance. It reduced the MAP evaluation on the validation data to 0.02922 and also a simultaneous decrease in the PrecisionAt500 on the test data by 20%. Generally, log compression was to handle the skewness problem towards large values. As we mentioned previously, almost half records in our sample data are with the count of 1. Despite the wide application, the log transformation would decrease the variabilities of data. However, we generally hoped to distinguish between likes and dislikes in the recommendation systems. This might cause a smaller difference in users' attitudes towards their preference than the original values. Thus, the underperformance was not surprising.

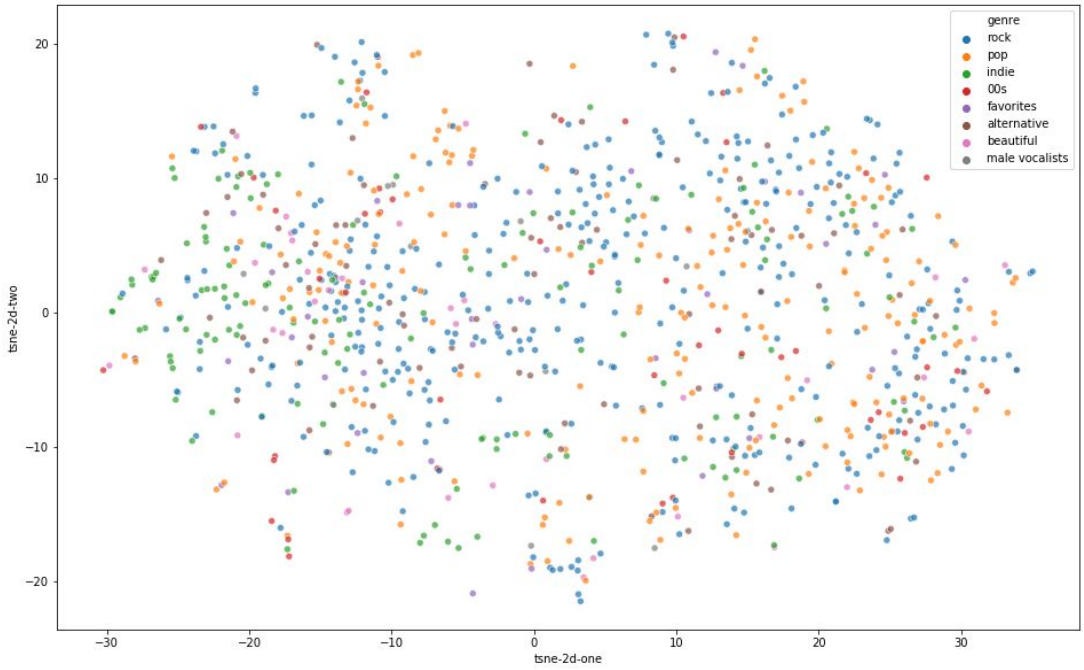
B. Exploration: T-SNE

Another extension we picked is Exploration where we visualized the learned embeddings with the additional datasets and see how the items were distributed across the learned space. In this particular example, we chose to visualize users by tagging their hidden variables with the genre of the song they like most. We used t-distributed Stochastic Neighbor Embedding (t-SNE) for visualization purposes.

The first step is to train the ALS model with the best parameter we got from hyperparameter tuning. After that, we exported the embeddings for each user to a dataframe of latent variables. The first dataframe contained two columns: (user_idx, latent variable). Then we searched the original dataset to find the track each user like most and save them as another dataframe. Then we joined tag.parquet with the favorite song dataframe on track_id. After merging, we possessed the second dataframe with (user_idx, track_id, genre) columns. Joining the first and second dataframe together creates the final dataframe for plotting with (latent variable, user_idx, track_id, genre) columns. Then we ran t-SNE with 40 complexity and 5000 epochs on the latent variable column. We used the first two generated dimensions by t-SNE along with the genre information, to plot the following graph.

In this graph, we chose to show the top 8 genre distributions. As we discovered, male vocalists usually appear on the top left while beautiful are on the left side of the graph. However,

some genres, like rock and pop, were too popular to observe a clustering for them. Another reason that t-SNE was not able to completely separate the data points was that those labels had too many noises in them and even the model cannot fully capture the variability. It was hard for t-SNE to cluster base on nebulous inputs from the model.



IV. Discussion: Big data on real-world application

In this project, we faced problems in the model building process, including, but not limited to large data storage, operation, and computation efficiency. Thus, we had to manage time wisely to finish all tasks in time.

Handling the random noises and scalability was the difficulty inherent in this type of practical application problem. Our model achieved an average ranking MAP of 0.03451 on the test set. The ALS model could not increase the performance significantly because of too many noises existing in data. We could see that same trend in the user visualization graph. The users who like rock music the best are evenly distributed across the manifold. It does not mean our model did a bad job at separating rock lovers from other genres, rather, it means that people love rock music the most love other music probably just as much. Randomness is a key aspect of human preference data because everyone is more or less a unique representation in the latent space. From modeling tuning perspective, modifying implicit rating feedback data is tricky as well. The modification strategy could quite hurt model performance if we have chosen improper methods.

Appendix - Hyper-Parameter Tuning Results

alpha	regParam	rank	performance
0.1	0.01	5	0.02608517883551579
1	0.01	5	0.02722834950134088
10	0.01	5	0.02486461317109056
0.1	0.1	5	0.02616847917320394
1	0.1	5	0.02728272320560010
10	0.1	5	0.02500017661577685
0.1	1	5	0.00007789948082169
1	1	5	0.01907174403697928
10	1	5	0.02416066105935739
0.1	0.01	10	0.02895572813159136
1	0.01	10	0.03087538648496492
10	0.01	10	0.03365692188734830
0.1	0.1	10	0.02975586902461756
1	0.1	10	0.03174539279707845
10	0.1	10	0.03396377324755474
0.1	1	10	0.00039074389818148
1	1	10	0.02609982708991549
10	1	10	0.03324246745786462
0.1	0.01	15	0.02895332104033823
1	0.01	15	0.03419895036348733
10	0.01	15	0.03760569416552457
0.1	0.1	15	0.03292977738737039
1	0.1	15	0.03469164143469834
10	0.1	15	0.03784141446271002
0.1	1	15	0.00051734098367510
1	1	15	0.02868420346354718
10	1	15	0.03762663535158089

Table 1: Hyper-parameter Tuning