BIG DATA PROJECT REPORT

Dieter Erben Vasconcelos (dev241)
Arushi Himatsingka (ah3243)

BACKGROUND

In this project, we try to build and evaluate a recommender system over a songs data set. We implement an ALS model, perform hyperparameter tuning and evaluate test metrics. We also try to add extensions to improve the performance of our model.

IMPLEMENTATION

Step 1: Creating Train Sample

One of the first tasks we worked on was creating a train sample. The reason for doing this was the size of the original train file. When trying to fit models or the StringIndexer, we were running into many memory issues. Therefore, we decided to create a train sample with the users for the validation and test sets, plus a subset of the data of the remaining users. In order to do this, we got a list of the unique users in either the test or the validation set. We made sure to get all training data on these users.

Along with this, in order to get some more data on other tracks these users had not listened to before, we got a 10% sample of the remaining data, where the user is not on the validation or test set but the tracks they listened to could and probably would be. We also acknowledged that while increasing our training sample we are not guaranteed to be getting tracks that will show up in the test and validation set, so our performance may/may not be better. The decision to go with 10% was a tradeoff we settled for in choosing between speeding up the running of the modeling, and keeping as much data as we could.

Step 2: Fitting String Indexer

Once we had our training sample created, we created a new dataset to train our StringIndexer. The StringIndexer inputs a set of strings and outputs unique indexes instead, making it more efficient to go through the data and reducing the size of the file. To train the StringIndexer, we selected the unique UserID's in our train sample mentioned above and all the TrackID's in the metadata. This provided the StringIndexer with all possible UserID's and TrackID's it could encounter in any of the datasets. We chose to go with the skip method of handling invalids, but based on how we trained the StringIndexer, there should not be any UserID's or TrackID's in the validation or test set that were not seen by the fitting of the StringIndexer.

Lastly, before fitting the StringIndexer, we decided to repartition our unique ID's sample mentioned above. We repartitioned based on the UserID column and used 5000 repartitions. Since these tasks were taking time and memory to run, we decided to separate them into different files. So for our first file, we saved the fit StringIndexer pipeline in order to use it in later steps of the process.

Following the creation of the StringIndexer pipeline, on our next file, we transformed the train sample to have indices instead of strings for the ID's.

For each ALS model we trained, we passed the parameters mentioned above for the current iteration, plus the UserID and TrackID and the User and Item columns respectively. For the rating column, we passed the count. We realize that the count is a method of implicit feedback, whereas rating is a method

of explicit feedback. Therefore, when evaluating the model and generating recommendations, we took this into account by choosing certain evaluation metrics, such as Ranking Metrics, that bide well with implicit feedback, using it as a sign of preference rather than of rating. Using RMSE would not have made sense here considering count is implicit feedback.

EVALUATION

We use Mean Average Precision(MAP) from Ranking Metrics from MLlib to test the performance of our model. MAP is a measure of how many of the recommended songs are in the set of true relevant songs on average, where the order of the recommendations is taken into account.

To tune the hyperparameters for the model, we set up a random search. As the number of hyperparameters keeps on increasing, computation required for Grid Search also increases exponentially. Thus, it is more efficient to randomly sample a combination of hyperparameters within the desired range. Also, when doing Grid Search, our pre-defined hyperparameters might miss the true optimal when multiple parameters are being simultaneously tuned[1]. Having limited storage and computation resources, we thought this would be efficient to find near optimal hyperparameters.

To implement this we set up a loop with 50 iterations which randomly chose a hyperparameter value for the rank, regularisation and alpha within a predefined range. We set the range as follows : rank between 10 to 100, the regularisation between 0.1 to 1 and alphas between 0.1-15.

RESULTS

Validation Results :

Based on our validation, our best MAP score was 0.0401, achieved from hyperparameter values of Rank = 78, Alpha = 14.287 and Regularisation Parameter = 0.4177. Below are some of our results for different parameters:

Rank	Alpha	Regularisation	MAP
11	0.34324418650093663	0.17951965088668903	0.01972345069050039
12	0.644263603585620	0.9461739215188891	0.02018401104795839
36	0.5870110425392082	0.9041575653609921	0.02197061002145755
55	0.43863083374425427	0.5358766963588152	0.023375412018558
10	1.039923922271615	0.20402518953264911	0.02424778666804966
72	0.6931458702847676	0.1252860664596362	0.02552504599359361
55	1.9795475696763031	0.8802621786880503	0.02716713138135346

[1] Bergstra, J. and Bengio, Y. (2012). *Random search for hyper-parameter optimization*. [online] Available at: http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

76	3.7324866040889185	0.3085624451464194	0.02746248709782712
65	0.9489500616928892	0.47703310828522905	0.02849585393585999
11	9.9631101314551751	0.9356821056355776	0.0294438919550919
14	2.17356873385478938	0.3425322198146118	0.03192665801677792
76	6.22972055305454217	0.37730202499604926	0.03239432097633995
61	3.367975368923951	0.316717284561213	0.03486524373971167
51	13.367975368923951	0.3009072845561213	0.03835234373175933
78	14.287069059772636	0.41772043857578584	0.04012232863195240 76

Test Set Results :
We implemented our best performing validation model with hyperparameter values of Rank = 78, Alpha = 14.287 and Regularisation Parameter = 0.4177 on the test set and got a MAP score of 0.038915. The MAP score for our test set is similar to our validation set, thus it does not seem like the model has been highly overfit.

EXTENSION
Once we got our basic recommender system running, with our optimal parameters based on our validation search, we decided to look into the different possible extensions for the model. We were interested in the way that we used count as a metric of preference.

Dropping low counts
First, we started with the suggestion of dropping low count values. For the low count values, we started by dropping all rows where the count was 1. This is because we are trying to see how much a user likes a specific song, and listening to a song once doesn't tell us much about preference, since it could have been played by accident, or the user may have not liked the song so that's why the user didn't play it again. After dropping these rows, and resampling accordingly, we ran our model with the tuned hyperparameters on the test set. Our test set performance was 0.037662. This does not seem to improve performance significantly, however, we must note that the when we drop all rows with count equal to 1, we reduce our data by half.

We considered dropping other counts greater than 1, yet when we inspected the data before, we realized that if we dropped these, we would be losing a considerate amount of data. If we dropped the rows with a count less than 2, we were losing 75% of our data. Therefore, we did not drop any counts greater than 1.

Log Compression
The next formulation we tried was log compression. Since we had a large range of possible values, we used log compression to reduce this range and see the effect on the accuracy of the model. We transformed each count by using the log SQL function (base 10), and used this modified count as our

implicit feedback. The results for this formulation gave a test MAP score of 0.038932, which is a minor improvement to our original performance.

Other Methods

Besides these two suggested formulations, we decided to research other ways to handle large ranges of possible values and handling implicit feedback in a better way. We decided to transform our data based on each user, rather than the entire range of values. Our first try was using z-scores, where we used a user's mean and standard deviation count and computed each z score for each respective track. This method gave us a couple of problems, since we were now getting negative counts and we had to consider the cases where the standard deviation was 0. Our second try was based on a paper we found comparing implicit and explicit feedback, and ways of transforming implicit feedback to improve model results.[2] Their way of normalizing data was to divide the each count value by the total number of artists a user listens to. This was done to take into account how much a user interacts with the system, and therefore standardize counts to each user. For our implementation, we decided to look into the ratio of count to the total number of tracks a user listens to, our proxy of user interaction with the system. As a user listens to more tracks, higher counts are penalized to reflect high interaction with the system. This ratio transformation resulted in a MAP score of 0.029431, which does not seem to improve our test performance.

CONCLUSION

Overall, through the project we got the opportunity to build a recommender system from scratch. We encountered numerous issues with the size of the data, as the cluster was having memory issues and we had to readjust the memory and sample our data. But, this was a useful exercise in understanding how to handle large-sized data.

CONTRIBUTIONS

Both team members worked together on understanding the project, brainstorming our methods and approach, selecting our extension and writing up the report. After that we divided up the implementation as follows :

Dieter : String Indexer, Train Sample, Extensions
Arushi : Validation and Ranking Metrics, Extensions

REPRODUCIBILITY

All our code can be accessed here : https://github.com/nyu-big-data/final-project-team

---

[2] Jawaheer, G., Szomszor, M., & Kostkova, P. (2010). Comparison of implicit and explicit feedback from an online music recommendation service. *Proceedings of the 1st International Workshop on Information Heterogeneity and Fusion in Recommender Systems - HetRec 10*. doi:10.1145/1869446.1869453