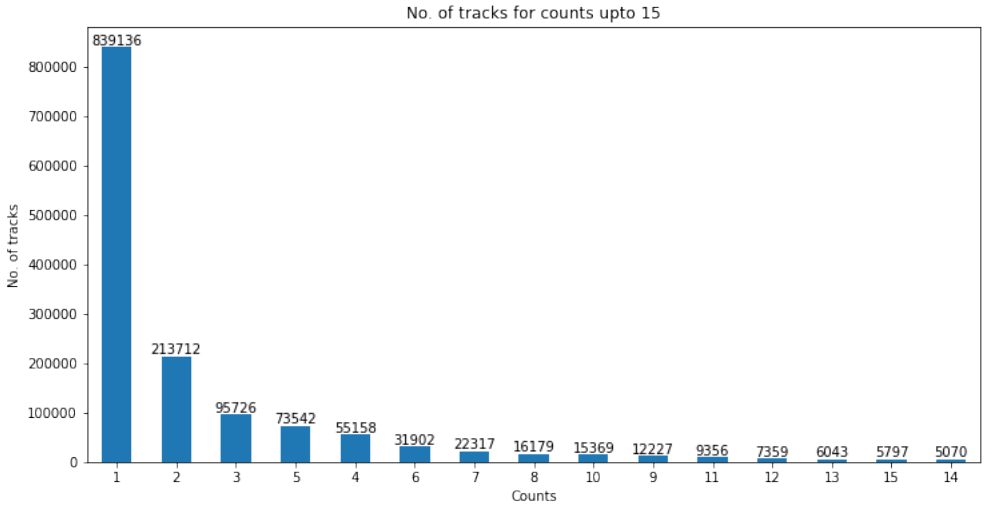
DSGA 1004 - Big Data
Final Project Report

The North Remembers
Chinmay Singhal (cs5597) and Sarthak Agarwal (sa5154)

Building recommendation systems (RS) using implicit feedback is a challenging task. Most of the literature on recommendation systems is mainly focused on modelling explicit feedback because of its convenient interpretation and the availability of well-defined evaluation metrics, such as RMSE, to gauge test-set performance. Nevertheless, certain ways are indeed available to extract confidence from implicit data and here we present and implement some of these methods to develop a RS using collaborative filtering for recommending songs to users based on the number of times (counts) a user listened to a particular song.

1 Implementation

The training dataset contains 1,110,000 users, each associated with a large number of tracks and corresponding count for each track. But the training set has only partial histories for the last 110,000 users with the rest of their histories present in the validation (10,000 users) and the test sets (100,000 users). So given the scale of the data, the first thing we did was to downsample the training dataset using an SQL query to only contain rows for those 110,000 users which are present in the validation and test set. This was done because eventually we are only going to recommend songs for these particular users and also a smaller dataset size gives room for faster implementation times. The bar chart below shows the number of tracks associated with counts 1 - 15 for the downsampled training dataset. The distribution is similar to a long tail distribution and the frequency for counts > 15 continues to decline rapidly.



The user and item identifiers in the datasets are string values but Spark's ALS model, which we are going to use for building the RS, requires numerical index representations. So we fit PySpark's String indexer function on the user column of the training set and transformed all the three datasets using it. Now, since the val and test sets have partial histories of users, there maybe some tracks in these sets which are not present in the training set. In order to include these tracks, we fit the String Indexer for the Track column on the 'Metadata' dataset which contains information for all the tracks, and then transformed the three datsasets using it. We then stored the three transformed datasets in parquet format without repartitioning. We used repartitioning as well with a partition size of 2000, but implementing ALS on the repartitioned datasets took a lifetime and so we did not go ahead with it. The file "index.py" contains the entire code for all the steps mentioned till now.

Now for evaluating the results after fitting the ALS model, we used Pyspark's RankingMetrics library for calculating the Mean Average Precision (MAP) score for each model. MAP requires two arguments, first is a list of recommendations for each user sorted according to the rank and the second is the ground truth label set for each user, which includes the track IDs the user has already listened to and the counts of which are already present in the datasets. We use the RecommendForUserSubset method to generate top 500 predictions for each user and use the pyspark.sql.functions library to extract just the list of recommend track IDs from the recommendations. We then join this list with the list of ground truth for each user and send this RDD to the RankingMetrics function for evaluation.

2 Evaluation results

We fit Spark's ALS model on the final downsampled and indexed training dataset, setting "maxIter = 10" and "implicitPrefs = True" because we are dealing with implicit feedback (counts). We used the validation data to tune the hyperparameters "regParam", "rank" and "alpha" for the ALS model. The range of values chosen for these hyperparameters were regParam = {0.01, 0.1,1,10,100}, rank = {20,50,100}, alpha = {0.01,20.0,40.0,100.0}. Running this validation approach took us almost a week as the jobs were getting killed most of the times and so we had to keep trying until the job went through.

We used a normal for loop for looping over the range of values and did not use Spark's built in cross-validation function because of two reasons. ne, the dataset is already large and there is no need for cross-validation over k folds and secondly we have our own separate validation set to test on and so there's no need to extract one from the training set, which is internally done by this function. We tried to incorporate as much variation as possible in selecting the range of values for each of the parameters. Some of the hyperparameter tuning results are presented in Table 1, with the MAP score rounded to the sixth decimal place. For looking at all the validation results, refer to the file "results.txt".

The best paramaters were found to be regParam = 10, rank = 100 and alpha = 40.0, with a MAP score of 0.054917 on the validation set. Using this set of parameters, the MAP score on the test set was found to be 0.055271, which we consider as our baseline model's accuracy. The file "crossval.py" contains the code for performing all the hyperparameter tuning and also for calculating the MAP score on the test set using the best parameter values.

RegParam	Rank	Alpha	MAP val set
0.01	20	20.0	0.034867
0.01	50	40.0	0.043984
0.01	100	40.0	0.050432
0.1	20	40.0	0.034757
0.1	50	100.0	0.042809
0.1	100	40.0	0.051215
1	20	20.0	0.036101
1	50	40.0	0.046382
1	100	40.0	0.054679
10	20	40.0	0.033212
10	50	100.0	0.049238
10	100	40.0	0.054916
100	20	100.0	0.017624
100	50	100.0	0.024186

Table 1: Valididation results.

3 Extensions

The extension that we implemented was 'Alternative model formulations'. We modified the count data for the training set and calculated the corresponding MAP values on the validation set for all the different modifications. To fit the model, we used only the best set of parameter values obtained previously and do not perform any hyperparameter tuning for evaluating these extension models due to resource limitations on Dumbo. The following modifications were tried -

1. Log compression with natural log: Modified the count data using a log transformation such that count = $\log_e$(1+count)
2. Log compression with log base 2: Modified the count data by using a log transformation with log base 2 such that count = $\log_2$(1+count). This ensures that counts that were already 1 still remain 1 after applying the log transformation while rest of the count values are reduced.
3. Squaring count: Modified the count data by squaring each of the count values. The reason was to see if increasing the relative distance between the counts affects the accuracy of the model.
4. Cubing count: Modified count data by cubing each of the count values. Same reason as for squaring counts.
5. Dropping low counts: Modified count data by first dropping rows with count = 1, then count <=2, then count <=3, then count <= 4 and finally count <=5. But this approach leads to addition of bias in the model as we are considering that if a user listens to a song less than or equal to 5 times, then the song is not too important for the user.
6. Binning counts: We binned the tracks having counts 1 to 5 as having count 1.

The results are presented in Table 2. We see that the log transformation with base 2 approach works the best and gives us the highest MAP value of 0.062177. On running this approach on the test set, using the same best parameter values, the MAP value obtained is about 0.062128 which is better than

Modification	MAP val
Log compression base e	0.059706
Log compression base 2	0.062177
Square count	0.040991
Cube count	0.024946
Drop count 1	0.040312
Drop count <= 2	0.032698
Drop count <=3	0.027348
Drop count <=4	0.023669
Drop count <=5	0.021249
Binning	0.050674

Table 2: Extension results

the base line model by a value of about 0.01 (baseline was 0.054679). This shows that log compression works much better than other modification strategies for handling implicit feedback.

The file "extension.py" contains code for the first four modification strategies. A fourth argument is required for running this file, which indicates the type of modification to apply - "log", "log2", "square" or "cube". The file "drop_count.py" contains code for implementing the drop count and binning modifications and this file also requires a fourth argument "drop" or "binning" for the applying the respective techniques.

Tuning the hyperparameters for each of the extension models could have further improved the test results as it is not necessary that the best basline model parameters would also be the best parameters for these different models. Given the constraints and limitations faced while running jobs on Dumbo, this analysis was the best we could do and there is many other heuristics that can be tried and implemented to further improve upon the presented recommendation system.

4 Contributions

The work was distributed equally among us and we discussed the ideas and implementation for each model before actually running in PySpark. The individual responsibilities were as follows -

Sarthak Agarwal (sa5154) -

- Implemented "index.py" for loading datasets, fitting string indexers and saving parquet files.
- Implemented "drop_count.py" for modifications 5 and 6, by dropping low counts and binning.
- Implemented "count_sorted.csv" which produces a csv file containing all the counts in the training set along with their frequencies.

Chinmay Singhal (cs5597) -

- Implemented "crossval.py" for fitting ALS models and for hyperparameter tuning on the val set.
- Implemented "extension.py" for modifications 1 to 4, which involve modifying counts using log compression with different bases, squaring counts and cubing counts.
- Generated the bar graph using count_sorted.csv and created the report.