DSGA-1004: Recommender Systems

Group 3: https://github.com/nyu-big-data/final-project-group-3

Yu-chen Chao
yc6371@nyu.edu

Shih-Lun Huang
sh7008@nyu.edu

Kathleena Inchoco
ki2130@nyu.edu

ABSTRACT

In this project, we implement a recommender system using data from ListenBrainz. We implement different recommender models to determine which one is able to produce better recommendations. We also analyze how long it takes to implement the LFM in different environments, i.e. Spark versus single-machine computation and see how spatial data structures can accelerate search at query time. The main focus of our analysis is the comparison between a baseline popularity model and the LFM. Ultimately, we find that the LFM produces better recommendations for users compared to the baseline popularity model.

CONTRIBUTIONS

Yu-chen: Implemented the pre-process and evaluation for the baseline popularity model and the LFM. Implemented the Annoy extension.

Shih-Lun: Implemented the non-voting case for the baseline popularity model and the training model for ALS. Implemented the Lenskit extension along with Kathleena.

Kathleena: Implemented the voting case of the baseline popularity model, helped refine the details of the ALS pre-process, and implemented the Lenskit extension along with Shih-Lun.

Everyone worked together to complete the report.

1 Pre-processing Data

We have one pre-process filter that removes any user who is in the 20th percentile or below of the number of tracks listened to per user. This means that these users and their interactions are removed from our analysis and do not exist in the train or validation set. They are also not reflected in our prediction of the top 100 songs for the baseline popularity model. Our reasoning is that users who are in the 20th percentile or below in terms of track count on ListenBrainz do not interact with the platform in a meaningful way such that we can make productive inferences about these users or extrapolate their interactions to productively generalize to the entire set of users.

2 Partition into Train and Validation Set

We partition the training set into train and validation at the track level within each user. We do this because we want the flexibility to recommend potentially new songs to a user and evaluate whether that song is a good recommendation or not. Note that this is different from splitting at the interaction level for each user. We don't want the same song to show up in the training and validation set because then the song is already known. If we split at the interaction level then we would potentially recommend songs that the user has already listened to already. In order to set up the partition into the train and validation set, we need to make sure that the data can be split on the songs. In order to do this, we randomly split the songs for each user into 5 potential groups. Groups 1-4 become the training set. Group 5 becomes the validation set. We do this by implementing a window function that assigns a random number of songs into a group while also grouping by the user_id.

3 Baseline Popularity Model

3.1 Model Variations

We implement three cases for the baseline popularity model. 1) Count the number of interactions for each song across all users 2) Implement a voting model without a dampening factor 3) Implement a voting model with a dampening factor. Since the first case is trivial, we will discuss the second and third case.

For the second case, we compute the top 100 songs for each user and assign a score (max of 100) based on rank for each song at the user level before we compute an average score for each song across all users. We did this because in the first case the model might be heavily biased toward people who listen to a lot of songs. To avoid this issue, we essentially rank songs based on the number of distinct listeners, but each user's "vote" is weighted differently based on the rank of that particular song for that user.

In the third case, we use the same voting model but introduce a more subtle dampening factor instead of dividing the average rank by the number of users who listened to the song. The computation of the adjusted score is as follows:

$$adj\_score_{msid} = \frac{\Sigma\, score_{msid}}{\Sigma\, listeners_{msid} + \beta}$$

Here, the beta is our hyperparameter. The reasoning here is that the beta term penalizes songs that have few listeners. In the case that a song already has many listeners, the bias term does not affect the score very much. However, when there are few users who listen to the song, the bias term affects the score greatly. We tuned our beta across a range of values from 1, 5, 10, 25, 50, 100. Our optimal beta is 50.

Note that for the voting model we remove a song if it is in the 5th percentile or below in terms of listener count per song. If a song is ranked well but doesn't have a lot of listeners we don't want it to skew our results to the detriment of songs that are also ranked very well and have many listeners.

3.2 Baseline Popularity Model Performance

After running all three cases of the baseline popularity model, we found that case 3 performed the best on the test set with a mean average precision (MAP) of 1.4366×10^{-4}. Similarly, the MAP of case 3 on the training set is 1.8495×10^{-3}. The MAP on the validation set was 1.8167×10^{-3}.

We observed that when we ran all cases of the baseline popularity model (whether using the small training set or the training set) that the MAP values on the test set were always smaller than the MAP values on the training set we used. Overall, the rankings are poor even when we try to filter out cases that are not representative of the entire population and even when adding a dampening factor.

4 Latent Factor Model (LFM)

4.1 Hyperparameters Setting

We implement the LFM from Spark and tune three hyperparameters: rank, regparam, and alpha. The rank defines the number of latent factors, regparam is the regularization term, and alpha is the weight of the implicit feedback.

$$min_{U,V} \sum_{(i,j)\in\Omega} c_{ij}(p_{ij} - < U_i, V_j >)^2$$

$$where\ p_{ij} = 1\ if\ R_{ij} > 0\ else\ 0,\ and\ c_{ij} = 1 + \alpha R_{ij}$$

We initially tried to tune the hyperparameters using rank in [100, 200, and 300], regparam in [0.01, 0.05, 0.1], and alpha in [10.0, 20.0, 30.0]. However, the cluster was overloaded with too many jobs to be able to train our model and evaluate results across all of these parameters. Therefore, we decided to run our analysis on a subset of hyperparameters in order to obtain timely results. Even though we only chose to run our model on only a few hyperparameters, we will discuss in depth how each hyperparameter affects the model performance.

It was specifically hard to train and evaluate the model with large rank because this increased the dimensionality of our utility matrix. Given the constraints on the dataproc, this was not feasible for us to run while also competing for resources with other students also trying to run their analyses at the same time. We limited the scope of our model to training on the small training data only.

The rank parameter defines how many latent variables we want to include for our feature vector. A larger value for rank means that there are more latent variables so we expect our model to produce better recommendations when the rank is higher. One of the challenges we encountered was getting our utility matrix to be a dense matrix. We tried using StringIndexer but it didn't work for us. The error we would get was 'No Space Left'. In order to resolve this, we made sure to reduce the rank size in our analysis.

The regparam parameter is the regularization parameter. The ideal reparam value will be neither too small nor too big. The regularization term is important because it helps our model to avoid overfitting on the training set in order to generalize to the test set. A regparam value that is too small will not prevent overfitting effectively. However, a regparam that is too large may not only be able to fit the training it may not generalize to the test set either. However, the ideal reparam value will come from tuning on the validation set.

The alpha parameter defines the weight we place on the implicit feedback. This variable is heavily dependent on the

data and our own discernment of validity of the implicit feedback. We should only be making the weight of the implicit feedback higher if we think that the implicit feedback is valid and useful to our analysis. Ultimately, this parameter needs to be tuned on the validation set in order to see which value is best aligned with our data.

The maxiter parameter is technically not a hyperparameter but it did have a large impact on the runtime of the model. It's important to make sure there are sufficient iterations of the model so that it runs to convergence. However, more iterations means more memory usage so this was challenging when competing for resources on the cluster. We decided to keep the maxiter relatively low so we would be able to compute results in a timely manner.

4.2 LFM Pipeline

Our analysis is developed across different scripts.

The script 'gen_id' creates the ids for user_id and track_id in both the training data and the test data. For the track, we first coalesce 'recording_mbid' with 'reconrding_msid' to get a unique identifier for each track. then we use row_number as the new track_id. Due to the LFM's dense index requirement, we only create ids for users and tracks that have appeared in the train_small interaction and test interaction tables.

The script 'ALS_pre_process' takes as input either the train set or the test set. If the input is the train set we do the train/validation split. The output at this step is a parquet file with 3 columns: user_id (INT), track_id (INT), score (INT). Note that the score is the interaction count, i.e. the number of times a user listened to a song.

The script 'ALS_train' generates a recommendation of 100 songs for each user which includes the user_id and the prediction for each user which is a list of 100 songs.

The script 'ALS_eval' takes in the recommendation list from ALS_train and compares it to the ground truth either from the validation set or test set and gives us a MAP score.

4.3 LFM Performance

Parameters	Rank: 1, regParam: 1, alpha: 10, iter=1	Rank: 10, regParam: 1, alpha: 10, iter=5	Rank: 30, regParam: 1, alpha: 10, iter=5
Train MAP	3.62E-5	1.63E-3	7.41E-3
Val MAP	1.48E-5	4.64E-4	1.21E-3
Test MAP	7.84E-6	4.07E-5	1.12E-4

Figure1. LFM MAP Results

After running all three cases of the LFM, we found that the last case performed the best on the test set with a mean average precision (MAP) of 1.12E-4. Similarly, the MAP of case three on the training set is 7.41E-3 and the MAP on the validation set is 1.21E-3.

The results reflect what we expect to see given the parameter settings. The higher the ranking, the more iterations we will need to converge and the results will be better than lower rankings when it converges.

5 **Lenskit Extension**

5.1 Lenskit Performance

This section compares the performance of a single machine and cluster configurations using the Alternating Least Squares (ALS) implicit feedback model. We implement the ImplicitMF model from the Lenskit library in Python for the single machine application. The parameters for this model -- 'features,' 'iterations,' 'reg,' and 'weight' -- correspond respectively to 'rank,' 'maxIter,' 'regParam,' and 'alpha' in PySpark's Latent Factor Model (LFM).

To effectively run Lenskit on our local machine, we first retrieve our preprocessed train and test interaction tables from the Hadoop Distributed File System (HDFS). This extraction process involves transferring files from HDFS to Dataproc, before pushing the parquet files to GitHub for easy downloading.

Following the download of the preprocessed parquet files, we execute Lenskit on our local machine. As Lenskit employs pandas dataframes, we convert all parquet files into this format. However, the HDFS output process

automatically splits these into multiple .snappy.parquet files. To successfully read these files, we utilize Dask dataframes, which combine multiple pandas dataframes. We then convert the Dask dataframe into a pandas dataframe.

Though the ImplicitMF model includes a prediction method, it only generates scores for a given user-item pair. To streamline this process, we implement a Recommender object that generates recommendations based on our predefined model. We then fit the training dataframe onto this Recommender object. The fitted Recommender is then wrapped with a batch.recommend method, generating 100 individual recommendations for each user based on their scores.

In the evaluation stage, we initially attempt to apply the precision_score method from sklearn. However, after reviewing the documentation, we note that its computation mechanism differs from PySpark's RankMetric. Consequently, we define a function to compute the mean average precision, equivalent to the MAP in RankMetric.

5.2 Lenskit Performance

We experiment with different parameter combinations based on results from the cluster usage. However, due to local machine memory limitations, we cannot output using high ranks or high iterations. Furthermore, we are only able to train on a smaller dataset instead of the full dataset. Below is the comparison result with rank of 3, regularization of 1, implicit weight of 10, and 3 iterations.

Implementation	Cluster	Single Machine
Fitting Time	70.48 seconds	232.25 seconds
Train MAP	2.23E-4	0.0
Val MAP	8.96E-5	4.94E-3
Test MAP	7.48E-6	6.17E-4

Figure2. Cluster and Single Machine Comparison

Our observations reveal that the cluster configuration outperforms the individual machine in terms of efficiency when fitting the model. It's crucial to highlight that the 0.0 MAP value for the training set on the single machine can be attributed to the Lenskit's Recommender object behavior. Specifically, it excludes training items during the recommendation generation, which results in this outcome.

Though we do not directly compare how data size might affect performance on clusters and a single machine, we anticipate that as data size increases, both time and memory usage will also increase for both models. While initially a single machine may outperform the clusters, we expect that the clusters will become more efficient after a certain data size.

6 **Annoy Extension**

6.1 Annoy Approximate Search

We use the Annoy library to implement accelerated search. Annoy is a C++ library with Python bindings to search for points in space that are close to a given query point. It is known for its query speed. The idea behind fast search is that the ALS solver recommends tracks for each user by brute-force-ly calculating the dot product of every track and each user and returning the tracks with top dot product values for that particular user. Annoy model leverages the spatial data structure to search the nearest neighbors. As a result, Annoy can speed up the search by only computing the dot product with the nearest neighbors instead of the whole collection of tracks.

6.2 Annoy Train

The Annoy training/inference process is described as follows. We first exported user factors and track factors from the ALS solver (with Rank: 10, regParam: 1, alpha: 10, iter=5) in spark. Then we built the AnnoyIndex with track factors. Lasly, we queried each user factor on the AnnoyIndex to get top 100 tracks.

6.3 Annoy Performance

Implementation	Spark ALS	Annoy()
Fitting Time	70.48 seconds	27.77 seconds
Train MAP	2.23E-4	2.37E-3
Val MAP	8.96E-5	1.51E-3
Test MAP	7.48E-6	4.63E-3

Figure3. Spark ALS and Annoy Method Comparison

As we can see, the result from Annoy model is better than spark ALS. This is contrary to our expectation as brute force search should have better performance than fast search. We reasoned that our LFM is not converged, so the MAP is worse, but we can already observed the efficiency gain from the fast search as fast search can yield better results within shorter time, meaning it will take spark ALS more time to achieve the same performance.

6.4 Annoy Hyperparameters Experiment

We further investigated how *n_tree* and *search_k* two parameters affects the Annoy model. *n_trees* is the number of trees in the spatial data structure, and *search_k* is the number of nodes to inspect during searching. We picked 2 *n_tree* values, namely 10 and 20, and caculated 2 *search_k* values with the following formula:

$$search_{k1} = ntree * \#\ of\ approximate\ nearest\ neighbors$$

$$search_{k2} = ntree * \#\ of\ approximate\ nearest\ neighbors * 5$$

# of approximate nearest neighbors is 100 here as we are querying top 100 tracks. $search_{k1}$ is simply the default *search_k* value, and $search_{k2}$ is five times of $search_{k1}$.

Time(s)	search_k=ntree* 100	search_k=ntree* 100*5
ntree=10	16.58 (+0%)	27.23 (**+64%**)
ntree=20	31.42 (**+90%**)	50.62 (**+200%**)
train MAP		
ntree=10	7.67E-4 (+0%)	2.37E-3 (**+209%**)
ntree=20	1.26E-3 (**+64%**)	1.49E-3 (**+94%**)
test MAP		
ntree=10	4.76E-3 (+0%)	4.63E-3 (**-3%**)
ntree=20	1.28E-3 (**-73%**)	8.22E-4 (**-83%**)

Figure4. Annoy Hyperparameter Experiments Results

We can observed that the larger *n_trees* and *search_k* are, the longer it takes because there are more tracks indexed and more tracks searched. Also, the larger *n_trees* and *search_k* are, the better train MAP we got because the data is better fit to the model. However, we also discovered that the Annoy model can be overfit easily as test MAP became worse as we increased the *n_trees* and *search_k*.

7 Conclusion

Overall, the LFM performs better than the baseline popularity model when producing recommendations for users. It performs better because it is more personalized and takes into account implicit feedback from the data.

In the first extension, we saw that it was still faster to implement the ALS model on the cluster than it was to compute on a single machine.

In the second extension, we found that Spark ASL does worse than the Annoy model due to the fact that our Spark ALS model has not converged because we only have 5 iterations.