Group 1 Final Project

Music Recommender System

GitHub: https://github.com/nyu-big-data/final-project-group-1/tree/main

Rod Aryan
Center for Data Science
New York University
NYC, NY, USA
ra2829@nyu.edu

Aryan Jain
Center for Data Science
New York University
NYC, NY, USA
aj3650@nyu.edu

George Zhou
Center for Data Science
New York University
Brooklyn, NY, USA
gz2214@nyu.edu

1 Introduction

In this final report, we present our group's comprehensive work on the Big Data final project. Our task was to construct and evaluate a collaborative-filter-based recommender system, utilizing the ListenBrainz dataset, which was divided into four key stages. The first stage revolved around developing a baseline recommendation system. This system, based on popularity metrics, which provided a fundamental framework to compare subsequent models. This popularity-based model served as our project's starting point, built using basic dataframe computations and optimized for performance on a validation set.

Next, we transitioned into the second stage, which involved implementing an advanced model using the Alternating Least Squares (ALS) matrix factorization technique. Spark's ALS method facilitated the learning of latent factor representations for users and items, with performance being further enhanced through hyperparameter tuning. The third stage of our project focused on the evaluation of both the baseline recommendation system and the ALS matrix implementation. We diligently assessed the accuracy of both models, making predictions and reporting scores for validation and test data based on Spark's ranking metrics.

In the fourth and final stage, we embarked on the exploration and implementation of two extensions to enrich our basic collaborative filtering model. These extensions, chosen from a variety of potential enhancements, were aimed at improving the accuracy and efficiency of our recommendation system. Throughout this report, we will elaborate on our methodologies, results, and each team member's contributions across these four stages, providing a holistic view of our project journey.

2 Data Processing

2.1 Splitting Into Sets

Into the early steps of our project, the process of partitioning our dataset into training and validation sets was a critical task. We aimed to create a robust split that would ensure the integrity and representativeness of our training and validation samples. To achieve this, we initiated a preliminary filtering stage, removing users with fewer than 10 interactions from our dataset. The rationale behind choosing a minimum of 10 interactions was guided by our intent to implement an 80-20 split for our train-validation setup. Therefore, with at least 10 interactions, we could ensure an 8:2 split ratio, providing a sufficient number of interactions for both training and validation purposes. We also attempted a filter of a minimum of 5 interactions but found that the 4:1 split from this filtering process was not as effective as the 10-interaction minimum strategy.

This user-centric approach to partitioning the data was designed to ensure that our models would be trained and validated on a comprehensive user base, encompassing the diversity of our dataset. By partitioning data at the level of individual user interactions, we safeguarded against the risk of excluding any user from the training set due to the randomness of the splitting process.

2.2 Key Creation

The key we used to identify songs was coalescing recording_msids and recording_mbids (if there was a recording_mbid for a given recording_msids). This allowed us to have a unique key for a given song. We also used this step to remove any duplicate songs if they shared the same recording_mbids.

While our primary model did not consider the metadata in key creation, we made numerous attempts to incorporate it into our recommender system. However, we faced significant challenges with these attempts. Our idea was to create a key based on the combined artist + track name, filtering the results by checking the cleaned keys with string similarity algorithms. This was in order to look for keys that may have been the same except for typos that occurred in data entry.

Using code with this goal in mind, we attempted a full Spark implementation, a combined Spark-Dask implementation, a full Dask implementation, and even manually increased the memory cap. Despite these efforts, we were unable to develop a code that would execute without timing out on the server. This challenge, while frustrating, provided valuable insights into the complexities of dealing with large-scale data and furthered our understanding of the computational limitations within such environments.

3 Model Creation

3.1 Evaluation Metric

In our project, we employ the Mean Average Precision (MAP) as a central metric for assessing the effectiveness of our recommender system. This measure provides a comprehensive view of how well our system is recommending the most relevant songs to each user, based on their individual listening histories. Our implementation of the MAP evaluation is realized through PySpark's RankingMetrics class, as demonstrated in the provided code.

First, we read the ground truth and recommendations data from our parquet files. We then select the 'recommendations' and 'ground_truth' columns, transforming them into a Resilient Distributed Dataset (RDD). This transformation is essential as the RankingMetrics class, which we utilize to compute the MAP score, operates on RDDs. Subsequently, we pass the RDD to the RankingMetrics class to generate a MAP score for the top 100 recommendations. This specific focus on the top 100 songs aligns with our project goal of delivering a limited, yet highly relevant set of song recommendations to each user. We debated and even ran tests for 100 and 200 songs, but found limiting the evaluation metric to 100 was more effective for our specific algorithms.

By calculating the MAP score, we gain a quantifiable way to assess the performance of our recommender system. This score provides us with an empirical foundation for comparing and improving our models and is instrumental in driving our efforts toward enhancing the overall performance of our music recommender system.

3.2 Popularity Baseline

Our project's baseline model employs a ranking strategy that aggregates the top 100 songs listened to by each user. The ranking is determined using a formula: ((sum of song rankings)/(# of rankings + beta)), which served as the basis for curating our recommendation set. To compute the sum of song rankings, we assigned an inverse value to the position of a song on a user's list of the top 100 most listened-to songs.

A crucial aspect of our model's development process involved experimenting with various hyper-parameters: the minimum number of songs listened to by a user, the training-validation split ratio, and the dampening value (beta) in our score calculation. We postulated that users with few listened songs might be considered as 'outliers', potentially skewing our model's performance. Hence, we focused on users with at least 10 tracks, which, in our observation, led to a significant improvement in our model's performance. In relation to the dampening variable, we tested values ranging from 0.85 to 10,000. Our experiments indicated that a beta value of 100 produced optimal results.

The training-validation split ratio was another influential factor that enabled us to control the bias-variance tradeoff effectively. We explored ranges for the minimum number of songs (5-10) for each user and various split ratios (80-20, 70-30, 90-10). Our decisions were guided by the Mean Average Precision (MAP) scores, leading us to settle on 10 as the minimum number of songs and an 80-20 split for training and validation sets. The final stage of our process involved comparing our list of 100 recommended songs with each user's top 100 most listened-to songs (or as many as they had listened to, up to 100). This final comparison helped validate the efficiency and relevance of our recommender system, providing us with a reliable measure of its performance.

3.3 ALS Model

The ALS recommender model works on matrix factorization and because of that is able to create more personalized recommendations than something like a popularity baseline. We first index our mbid-msid keys into integer values by first merging them together using the *coalesce* function to ensure we had a unique key. We then indexed the unique

keys using SQL. Building the ALS model is fairly straightforward after that but the main crux of the problem is hyperparameter tuning, which is performed on the validation set. We mainly look at 4 hyperparameters that we look to tune (rank, maxIter, regParam, alpha) and we also set ImplicitPrefs to True as interactions are more so implicit feedback than it is explicit (which something like ratings would have been).

- Rank: This is something that increases computation time as it increases while also increasing MAP scores. The ideal rank is where MAP score starts to decrease after continuously increasing as after that rank, we tend to overfit to the training data. However, due to cluster restrictions, finding this "ideal" rank was hard so we turned to the elbow method where we choose the rank after which we don't see huge changes in the MAP score. The rank chosen is 100.
- maxIter: This according to our understanding refers to the iterations for which the process runs. The ideal maxIter would be where we observe convergence in MAP scores. We find this to be 15, this is not the ideal convergence (0 change from the previous iteration) but more so has very small changes after that.
- regParam: This focuses on controlling the L2 regularization. We look at the values [0.01,0.1,1,2,5] and find 2 to be ideal for our dataset. This higher penalty ensures that we don't overfit to our training data which is very important in our case as there is a difference in time between the training and testing data. We find the regParam to be 2.
- alpha: This focuses on controlling the elasticnet regularization. Very similar to regParam defined above. We find the ideal alpha to be 1.

We find that the time taken for .fit and .recommendForAllUsers (which are the 2 computationally expensive processes) takes 18.4 minutes.

4 Results

Upon completion of our modeling process and thorough testing on different data sets (training, validation, and testing), we obtained the following results:

For the single machine model, we achieved a Mean Average Precision (MAP) score of 0.258831333729467 on the training set, 0.2086554554809 on the validation set, and 0.0112283846046 on the test set.

The Nearest Neighbor model, a high-performing model, yielded MAP scores of 0.997821285448928, 0.9590913044018973, and 0.024543061849931765 on the training, validation, and test sets, respectively.

The ALS model produced a MAP score of 0.416222 on the training set, 0.37819 on the validation set, and 0.01617741765332664 on the test set.

Our Popularity Model, when applied with a damping factor of B=100, achieved MAP scores of 0.0008408899485992566 on the training set, 0.001026558166159579 on the validation set, and 0.00012578 on the test set. A lower damping factor of B=0.85 yielded MAP scores of 6.393726709116846e-05 and 0.0002269237081517996 on the training and validation sets, respectively. Without the damping factor, the test set achieved a MAP score of 6.275050442508558e-05.

To visualize the performance of each model, we can look at the following bar graph. The graph displays the MAP scores of each model for the training, validation, and testing sets, thereby providing a clear comparative overview of their performance.

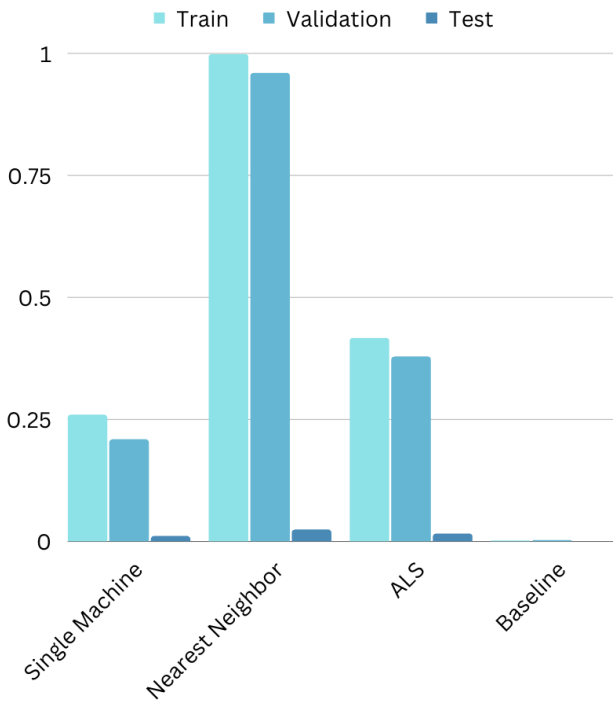

Figure 1

We see that the testing set performs considerably lower than the other 2 sets, which we attribute to one major factor: the difference in time. There are a significant amount of new songs that get popular every year which we could not

incorporate into our dataset. As per our learning, we believe that all the big players use previous user data and metadata of tracks to recommend new songs, something we did not have access to in the interactions dataset. The graph provides a clear visual representation of the relative performance of our different models across the three datasets. This comparison aids in understanding the strengths and weaknesses of each model, guiding further optimization and improvement efforts.

5 Extensions

5.1 Single Machine Implementation

For the single machine implementation we looked at both lightfm and lenskit. Lightfm required the construction of a sparse (coo_matrix) matrix, which we faced some issues with. Based on these difficulties, we moved to a lenskit implementation, whose install instructions can be found in the requirements.txt file within our repository. The biggest challenge was getting our preprocessed data onto our local machines. To go over this, we download the parquet files as snappy.parquet. Coming to the implementation we concat all the snappy(s) into a Dask dataframe. This leads to a Dask framework that is ideal for single-machine implementation.

As we see through the results mentioned previously, we achieved numbers similar to the ALS implementation. The time taken is 57 minutes for the ".fit and .recommend" process. We observe that this takes more time than the ALS model. This helps us make the inference that for this dataset a single-machine implementation is not better than a cluster implementation. This makes sense when we look at the size of the dataset, which is pretty sizable for local machines.

5.2 Nearest Neighbor Algorithm

The main challenge was transforming our data set into a sparse mat where each row is a user, and the columns are features of the user. The function we used to transform the training data was *CountVectorizer*, and then used *Normalizer*, so that each row represents a binary vector indicating the presence or absence of songs for a given user. To do the actual LSH, we used the *BucketedRandomProjectionLSH* function that was available in pyspark. Similarity between users was based on cosine similarity. The hyperparameter tools we fine-tuned are *bucketLength*, *numHashTables*, and *threshold*. Higher *bucketLength and numHashTables* meant longer but more accurate computations.

The threshold controlled how many users we would consider similar enough. In the end, we decided on values of 1, 5, and 0.5 for *bucketLength*, *numHashTables*, and *threshold* respectively. For each user and their similar users list, we recommended the top 100 songs the similar users listened to, based on the aggregated listening count. We preemptively removed songs that the user already listened to. The implementation was that of a "user-based model".

6 Contributions

In the successful execution of this project, the collaborative efforts of all group members were pivotal. The entire team worked cohesively on the initial data processing, with Aryan's ingenuity coming through for the team as he recommended and implemented the various filtering options we discussed. George's leadership significantly enhanced the process of developing the popularity baseline, creating the popularity metric both with and without the damping factor. Rod assumed responsibility for implementing the Mean Average Precision (MAP) as the core evaluation metric, contributing greatly to the standardization of our models' assessment.

Further, Rod and Aryan pooled their resources to work on the ALS portion, with Aryan taking the lead in developing the single-machine extension. Simultaneously, George spearheaded the nearest neighbor extension, contributing an additional dimension of complexity and sophistication to our recommender system.

Finally, Rod's organizational skills and meticulous attention to detail were instrumental in consolidating our work and finalizing this comprehensive report. The synergy of this group has resulted in a robust, comprehensive, and insightful project.

7 Conclusion

In conclusion, our collaborative-filter-based recommender system, constructed and evaluated on the ListenBrainz dataset, has demonstrated a range of successful outcomes, with some areas identified for further improvement. Through careful data processing, model creation, and iterative testing, our team has been able to develop and refine a system that can deliver personalized music recommendations to users.

The systematic partitioning of data into training, validation, and test sets, coupled with the strategic implementation of key creation and evaluation metrics, have been integral to

our project's robustness. Our project saw the implementation of a baseline popularity model, an advanced ALS model, a single-machine implementation, and a nearest-neighbor algorithm. The comparative analysis of these models, as visualized in our bar graph, has allowed us to gauge the effectiveness of each approach, its strengths and weaknesses, and the potential avenues for future enhancement. The challenges encountered, particularly in incorporating metadata into our recommender system and handling large-scale data, have provided valuable learning experiences.

Notwithstanding these challenges, the project has been successful in delivering a high-performing recommender system, as evidenced by the Mean Average Precision (MAP) scores.

8 References

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.recommendation.ALS.html#pyspark.ml.recommendation.ALS.implicitPrefs
https://stackoverflow.com/
https://lkpy.readthedocs.io/en/stable/index.html
https://chat.openai.com/
https://towardsdatascience.com/recommendation-system-in-python-lightfm-61c85010ce17