Yahoo Music Recommendations via Weighted-Average Ensembling of Classifiers over User Review History

Alexander Pellegrino
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Turnersville, United States of America
apellegr1@stevens.edu

***Abstract*— Machine Learning is often a matter of balancing trade-offs - Computational Cost versus Accuracy. Specificity versus Generalization. Scalability to Larger Datasets versus Precision for Smaller Ones. This paper focuses on the foremost of these, detailing several implementations of popular classifiers such as Random Forest, Logistic Regression, and Factorization Machines over a large dataset making use of Apache Spark. In addition, a deeper analysis of the data is performed utilizing an optimized brute-force search algorithm. The project aims to predict user preferences for new music based upon their past reviews while balancing the time needed to process the data with the accuracy of the final result. In the end, a weighted average ensemble is calculated based on the prior outputs and their known accuracies in an attempt to improve the overall result while minimizing additional computation.**

***Index terms*—Ensemble Learning, Big Data Analytics, Weighted Average Methods, Apache Spark (PySpark), Machine Learning Algorithms, Performance Analysis**

I. Introduction

Music recommendation is by no means a new subject. For almost as long as the music industry has existed there have been ways for listeners to receive recommendations on what to listen to. What's the most popular, suggestions from friends and family, or simply asking a seller what they like. However, modern music recommendation systems utilized by streaming platforms such as Apple Music, Spotify, and YouTube Music must process massive throughput on a day-to-day basis in order to maintain their regular business operations. This dataset can be massive, on the order of hundreds of millions of user requests per month[1], so having the ability to process a user's history and quickly produce a high-quality suggestion based on it is crucial to success. This project aims to do just that, examining the trade-offs between computational time and suggestion quality based on a known but hidden set of ground truth values. The original training dataset is provided in the form of a list of user ratings for different tracks (songs), genres, artists, and albums. Accompanying this is a list of track IDs with information about the corresponding artist IDs and album IDs when known, as well as any flagged genre IDs. This allows for the comparison of tracks that have the potential to be recommended to the user against the similarity in defining features to ones that they have reviewed prior. Through this it is possible to determine which tracks are most likely to be similar to ones the user has rated highly and which are most likely to be similar to ones the user has rated poorly. (With the inverse also being true, in that we know which tracks are dissimilar to those the user has given a high or poor rating to.) These two key components form the basis for building a classifier system over a set of tracks which can determine what songs from that set the user is most likely to rate highly, and thus the ones that should be recommended to the user first. It can also prune out tracks the user is very likely to rate poorly, ensuring that they never show up in the user's recommendations in an ideal world.

II. Methods

Prior to use in any of the classifier approaches, the dataset needed to be cleaned up. As some tracks lack artist or album information, and most have a dissimilar number of genres compared to one another, the raw dataframes are filled with holes which are then converted by **pandas** to NaN (not-a-number) values. This isn't inherently problematic in and of itself, but did necessitate the creation of functions which can safely parse the values of cells with robust error handling. These functions were placed into isolated Jupyter code cells that could be copied for reuse throughout each classifier implementation moving forwards.

The first classifier created based upon the aforementioned dataset took a naïve brute-force approach, defining functions to compare the similarity between two tracks via Jaccard Similarity and to estimate a user score based upon their prior reviews. The score was calculated via the summation of the rated score multiplied by the percent similarity between the track's properties. Initially this was done using Cosine Similarity,

however due to the vastly different number of genres labeled for each track often added a tremendous amount of bias. (For example, with Cosine Similarity, a track that was tagged as Rock and Metal would be considered dissimilar to one marked Rock, Metal, and Punk since they have a differing number of categories, despite having similar tags.) Once this switch was made, the program was then designed such that for each user it would compare each of the potential track suggestion candidates against every entry the user rated in the past, then suggest the top scoring half and decline suggesting the bottom scoring half. Unfortunately, while able to capture a great amount of detail, this purely brute-force approach proved far too slow to be feasible on a dataset of this size, taking over 6 hours to run fully for just 120,000 suggestions to 20,000 users on high-end hardware. In order to speed things up, the library was swapped for one more heavily optimized for performance, the dataset was pruned on a per-user basis, and the underlying program logic was simplified.

Firstly, the popular **pandas** library was swapped for the newer **polars** library, coded in Rust[2]. This library offers significant performance benefits for inline dataframe operations, at the cost of worse interoperability with other libraries. Fortunately, interoperability was rarely utilized throughout the program, so very little needed to be changed regarding the underlying structure. Secondly, the classifier was redesigned to first find all the unique user IDs in the target output set, then combine these into a native python list, as enumeration through a list is faster than through a dataframe. From this list, the speed of **polars** could be used to quickly create pruned dataframes with only the relevant suggestions for each user contained within. These simplified frames were then used as rapid lookup tables, rather than for iteration through. For each track suggestion candidate for the user, the album and artist were extracted. It was then checked whether the new user-specific table contained a rating for that album or artist or not. If it did, then the rating for the track was scored by adding one half of the user's rating for each. (For example, a song where the artist was rated 80 but the album was left unrated would score a 40, whereas a song where the artist and album were both rated 70 would score a 70.) At this point the program's prior behavior was left intact, suggesting the top half of the scores to the user. This new approach resulted in a program that was still fairly slow, taking around 20-30 minutes to run on average, while simultaneously discarding all consideration of the genres as a factor. Despite this, it proved to be highly accurate, scoring 84.533% accuracy against the ground truth. Had genres been taken into consideration this result would likely have been pushed even higher, however due to the complexity of the implementation and the drastic resultant increase in runtime genres were not deemed a worthy edition, as near 85% accuracy is already rather significant.

The second classifier focused far more on improving speed, making use of GPU acceleration via the Apache Spark library[3]'s Python bindings. (**PySpark**). Much of the code for this classifier was provided already as a sample by Dr. Rensheng Wang of the Stevens Institute of Technology School of Engineering and Science, and made use of the Alternating Least Squares (ALS) Matrix Factorization implementation included in the library. Run locally on the NVIDIA RTX 4090 GPU, this implementation showed a significant performance gain over the prior naïve brute-force implementation, outputting a prediction set in mere seconds. However, this implementation was not without fault. Primarily, the accuracy of this implementation, despite including all the dataset (album, artist, track, and genre ratings) was considerably lower than the initial approach, achieving an accuracy of just 53.516%. In addition to this, some outputs were lost during Matrix Factorization, necessitating implementation of a merge function which would find the missing outputs and use their values from the prior implementation's results to get a complete set of predictions.

The remaining four classifiers, those being Random Forest, Naïve-Bayes Prediction, Logistic Regression, and Factorization Machines, were all also trained making use of the default implementations in Spark. However, they made use of a different set of training data to the initial attempts. Firstly, the program parsed the training data using the existing parser, creating the classic frame of track, genre, artist, and album ratings. However, for this final section, a small subset of the ground-truth values were provided for training. These values were added as a new column to the dataset, and given precedence in training such that the recommended songs could be identified as "solidly liked" and the non-recommended ones as "solidly disliked." Once this new training set was ready it was stored inside a **pandas** dataframe. Individual functions were then created for each classifier, taking the training dataframe as a parameter and returning the classifications they came up with as a new one. These functions were then called one at a time, each saving their predictions to a correspondingly named CSV file for output. These classifiers however, despite the additional information available to them, yielded similar results to the prior **PySpark** classifier, with accuracies ranging from 49.977% to 50.168%, within margin of error of one another. Their primary advantage, once more, was in their speed, each classifier taking only seconds to train.

Finally, in an attempt to improve the overall classification, the predictions from every classifier, including the naïve brute-force classifier, ALS classifier, and standard **PySpark** set of Random Forest, Naïve-Bayes, Logistic Regression, and Factorization machines were all combined using weighted-average ensembling. In effect, each classifier's classification for each data track, either 0 or 1, was multiplied by its accuracy

(such as a song rated 1 being multiplied by 0.5003 for the Factorization Machines classifier due to its 50.03% accuracy), and a sum was taken for each suggestion. Whichever half of the suggestions for each user received the most cumulative points received the 1s for "recommend", while the bottom half received 0s for "do not recommend." Unfortunately, the result yielded no considerable improvement, in fact showing diminishing returns with the inclusion of the naïve brute-force classifier result. When the results of that classifier were included, the accuracy was reduced to 73.089% (down from 84.533%) due to the less accurate classifiers outvoting it with incorrect predictions. When it was excluded, the ensemble provided no significant change in accuracy, likely due to the fact that all the other classifiers had predictions within 1% of one another, leaving results around 50%.

III. Results and Discussion

The first classifier, involving a naïve brute-force technique with enough optimization to make running it potentially feasible, yielded the highest accuracy by a considerable margin, at 84.533%. However, it was also the slowest to run by multiple orders of magnitude, taking 20-30 minutes per run on average while every other classifier tested ran in just seconds. This approach shows great promise for a dataset of this size with further optimization to speed up the predictions and/or the integration of the genres as a factor.

The second classifier, involving the Alternating Least Squares method, yielded middling accuracy of 53.636%. However, it was incredibly fast, able to train and make predictions faster than it could read the data from the disk on the test machine, leaving the few seconds it took to load the CSV file into memory as the bottlenecking factor. With further refinement to hyperparameters and better organization of the data it could prove a highly useful tool for rapidly making accurate predictions. As it stands though, the accuracy is fairly low and may lead to some user complaints.

The third classifier, involving the Random Forest method, yielded the lowest accuracy of any tested classifier, requiring careful tuning just to bring accuracy to 49.977%. It offered no significant speed difference from the Alternating Least Squares classifier tested prior, again bottlenecked mostly by the time to load the data from the disk into memory on the test system, while simultaneously being more challenging to implement and tune. It is possible there was some user error involved in using it, however based on these preliminary results it does not seem well suited for this particular dataset.

The fourth classifier, involving the Naïve-Bayes method, yielded again middling accuracy at 50.168%. It offered no considerable differences in speed relative to either of the aforementioned **PySpark**-based classifiers, however required no modification to the default settings to outperform the Random Forest approach. For this particular dataset **PySpark**'s Naïve-Bayes implementation seems particularly suited to creating a model that's very quick and simple to implement, runs fast, and has decent results, though still nowhere near the initial brute-force method.

The fifth classifier, involving the Logistic Regression method, yielded precisely the same accuracy as the Naïve-Bayes classifier above at 50.168%. This is interesting to observe, as these two actually did output vastly different results, making a total of 15,013 different predictions of the 120,000 outputs from each classifier. This is a 12.51% difference in the output predictions, yet yields a less than 0.001% difference in net accuracy. Still, used individually, this only shows that the two classifiers are interchangeable in this particular context.

The sixth and final classifier, involving the Factorization Machines method, yielded another middling result of 50.03%. For similar reasons to the prior **PySpark** classifiers, it provides no significant new information, bottlenecked by the stage of loading the data into memory rather than processing time and giving no significant difference in output accuracy, even when ensembled with the other classifiers.

The weighted-average ensembling of the classifiers implemented throughout the semester did not prove helpful in this particular case, however the fact that it provided no noticeable improvement when used across multiple **PySpark**-based classifiers indicates the potential for user error in the implementations, such as bad settings or introduction of bias. This is most heavily indicated by the lack of significant change in classification accuracy, marking that despite all five **PySpark** classifiers using different methods, they all output similar enough results that ensembling them together made few modifications to the final output. Only when including the naïve brute-force classifier, which was implemented from scratch, was any considerable difference visible in the final resulting accuracy. This difference, however, involved the reduction of accuracy of the more accurate classifier by ensembling it with lower accuracy ones, rather than a mix of varying results providing increased accuracy as intended.

IV. Future Considerations

Moving forwards, it could prove interesting to investigate the pairing of the Naïve-Bayes classifier and the Logistic Regression classifier as implemented earlier. Despite having considerably different outputs in terms of the actual predictions made, they yielded precisely the same accuracy, or at least within the 0.001% precision provided by the evaluation software. A different ensembling approach would be needed, as currently they have the same voting weight so couldn't influence one another, however due to their comparable accuracy but different results they should theoretically show a significant difference in final accuracy, for better or for worse, if

combined. Unfortunately this detail was not picked up on during testing and any difference that would have been made between the two was buried under the weights of the other classifiers.

V. Conclusion

The various methods of classification implemented over the course of the semester for this project showed a tradeoff between processing time and accuracy to a tremendous degree, with an increase in computational time by several orders of magnitude being the only classifier implementation to show a significant spike in accuracy. However, human error is a likely culprit in this particular result due to the minimal variance between classifier outputs throughout the course. Only a from-scratch implementation showed any significant improvement in accuracy over the "off the shelf" ones, but the liklihood of a solo researcher with little experience in the field creating a better implementation than industry standards is highly unlikely. More likely there is a mistake involved in the data preparation or training of the "off the shelf" **PySpark** classifiers, resulting in their similar outputs. Weighted-average ensembling should help settle disputes between classifiers in theory, but for this particular project there were so few disputes to begin with that it made no considerable impact when applied to the majority of implementations, with a negative one when appying those implementations to the high-accuracy one. This most goes to show that proper handling and preprocessing of data is just as important, if not more so, as the actual underlying algorithm making predictions from that data.

References

[1] Brian Dean, "Spotify User Stats (Updated March 2023)". [Online]. Available: https://backlinko.com/spotify-users#spotify-monthly-active-users

[2] "Polars: DataFrames for the new era". [Online]. Available: https://pola.rs/

[3] "Apache Spark: Unified engine for large-scale data analytics". [Online]. Available: https://spark.apache.org/