EE627 Music Recommender Final Project

Team Name: Insert Team Name Here

Team Members: David Krauthamer and Bradley O’Connell

Summary of Methods Used and Their Performance

Naive Brute Force Search

The first method we attempted to use was a brute force searching method. We were originally supplied with the following files:

- testItem2.txt: Test data set containing user ids followed by track ids to determine ratings for.
- trainItem2.txt: Training data consisting of user ids followed by a list of track ids, album ids, artist ids, and genre ids with corresponding ratings for the given ids.
- trackData2.txt: Hierarchical data for all known track ids detailing what album, artist, and potential genres a song belongs to.
- albumData2.txt: Same idea as previous file, but for albums and their corresponding artists and genres.
- artistData2.txt: A list of all artist ids
- genreData2.txt: A list of all genre ids

The first step of this search was to parse all of the supplied data into something we could work with in Python. All code related to parsing for the brute force search is contained in the “parsing.py” file.

The first important file we parsed was trackData2.txt, which was parsed into a dictionary of track id strings mapped to TrackEntry classes, which contained the corresponding album id, artist id, and list of genres assuming that they existed for that particular track. This mapping made it very easy and efficient to access information about a track, which we needed to do often when trying to decide which tracks to recommend to the user. As for the album, artist, and genre data, we did parse these into Python data structures, but they inevitably went unused because the test data was only given as track ids, so there was no need for these other hierarchies to be present.

The second important file was trainItem2.txt, which was parsed into a dictionary of user id strings to dictionaries of ids (track, album, etc.) to ratings. This dictionary of dictionaries was able to represent the entire history of ratings for every user of the music service.

The final file to be parsed was testItem2.txt, which was parsed into a dictionary of user ids mapped to a list of track ids. This could then be easily iterated through to get each user id and set of tracks we needed to decide on recommendations for.

Each of these parsed dictionaries / structures was very large, and took significant time to parse into something usable. Our solution to this was to use the built-in pickle module of python, which allowed us to serialize and deserialize the parsed structures, avoiding having to compute them every time the program was run.

With all of this parsing out of the way, we were now able to begin performing a search and using the collected data in some way. For each of the methods of calculating a score for a particular track, we used the same general search method. The search worked as follows:

- For each user id and set of tracks,
    - For each track in the set of tracks
        - Get the corresponding TrackEntry class (containing the album id, artist id, and potential genres) from the track hierarchy map.
        - Get the user’s rating history from the user rating history map
        - For each part of the track’s hierarchy, check whether the user has rated that thing, and if so do something with that rating
        - Potentially do more processing after checking for each possible thing the user has or hasn’t rated, then return a score for that track.
    - Sort the list of scores we’ve made for each track
    - Assign the bottom three a value of zero, and the top three a value of 1, then add this to the submission csv file

Now we can discuss the various different scoring methods we tried as a part of the naive search method. Each of these methods was implemented as a function within the “search.py,” so check there for the exact implementations.

Plain Average

The first type of average we tried was a plain and simple average of all the ratings given by the user. This particular method had one of the lowest performances, due to the many shortcomings it has. By taking only a plain average every rating the user has made is considered equally. If the user had rated that exact track in the past, that would be considered as equally as them having rated a genre of the track. Obviously a genre is one of the most distantly related things to a track, and things like a track, album, or artist rating should be given more importance.

Weighted Average

The second and most experimented with method was using a weighted average to generate a score. The weights assigned to each possible rating the user could have made looked something like this:

- The exact track has been rated in the past: highest weight
- The album of the track has been rated in the past: second or third highest weight

- The artist of the track has been rated in the past: second or third highest weight
- A genre of the track has been rated in the past: lowest weight
- How many of each type of rating there were was also used to adjust the weights

We tried many different combinations of weights in an attempt to maximize the score we received from kaggle. The benefit of a weighted average is we could control how much a particular variable was worth. For example, if somebody has rated an exact track, then it makes sense to give that a significant weight in terms of whether or not we recommend it to them. Conversely, if somebody has only rated the genre's related to a track, those should have less impact because there can be many of them (a song may have many genres), and they are the least closely related metric we have about a song. Using a weighted average delivered better performance compared to a plain average, and compared to most of the other methods (including ML classification techniques) it tended to perform better or slightly worse while taking less time for analysis to complete. This may be partly attributed to the caching code we implemented for the naive brute force search as a whole.

Non-linear Weighted Average

In addition to weighing each linear datapoint separately, we also attempted to non-linearly combine these data points. The function we found most useful for this was taking the square root of the ratings, this more heavily differentiated low and high ratings while "smoothing" out the highest ratings and making them more similar and thus more reliant on the weights. This particular method actually produced the highest scores besides the ensemble method, even scoring better than the ML Classification methods. We also attempted to raise the data points to a power higher than 1 (instead of taking the square root. However, in making lower ratings more similar and higher values more varied, we achieved worse results in our model.

Total Sum

This method is fairly self explanatory, we simply summed the ratings the user had given to anything related to a particular track, and used that as a track's score. Similarly to the plain average method, this had the problem of being biased towards genres, and having no way to control how important some variables were compared to others. A pure sum also introduces a bias towards the number of ratings, where having a larger number of ratings means the score will tend to be higher, even if the ratings being added up aren't necessarily all high. As an example, if the user gave a track itself a rating of 90, but the album and artist of another track a rating of 60 for each, the second track would have a higher score, and be more likely to be recommended to the user, even if the ratings given weren't good.

PySpark Matrix Factorization

The second method we attempted to use was Matrix Factorization. The general idea behind matrix factorization is to create a massive sparse matrix which represents the data we have in the system, and use mathematical techniques to try and fill in the gaps of what is missing. For our recommender system each row represented a single user, and each column was a possible track, album, artist, or genre ID they could have rated. To generalize the math that matrix factorization performs, when it comes across something that a user hasn't rated, it will attempt to find other users with similar tastes and ratings that have rated the missing one, and make an educated guess based on how these other users reacted. The implementation of this can be seen in the “matrix_factorization.pdf” file, which is a pdf output of the Google Colab notebook that was used for homework 8. After performing matrix factorization on our dataset, we found that it performed extremely poorly, giving us our worst performance on kaggle by far except for a submission for a different method where the ones and zeroes had been flipped on accident. Our best guess as to why this performed so poorly is because of how sparse the matrix itself was. From the input data there were roughly 49 thousand users with an average of 252 ratings each. There were around 295 thousand individual track, album, artist, and genre ids combined, so each user rated roughly 0.08 percent of the total data set. Another possible reason for the poor performance could be not having enough users. The more users there are, the more likely it is that some of them have similar preferences, and can be used to make predictions about other users like them.

PySpark ML Classification

Gradient-Boosted Trees

We also used a gradient-boosted tree (GBT) classifier to predict the recommendation of tracks for different users based on their previous ratings. A GBT classifier is a supervised machine learning model that combines multiple weak decision tree classifiers (DTCs) into a strong one by iteratively fitting them on the residuals of the previous DTCs and using a weighted sum of their predictions. A residual is the difference between the actual and predicted value of the target variable. By fitting on the residuals, the GBT classifier tries to correct the errors made by the previous DTCs and improve the accuracy. A weighted sum of predictions means that each DTC has a weight that reflects its contribution to the final prediction. We implemented the GBT using the Python scikit-learn library. In terms of performance, GBT performed the best out of the ML classifications we tried, but still performed slightly worse than our best Naive Search scoring algorithm.

Random Forest

We used a random forest classifier (RFC) to predict the recommendation of tracks for different users based on their previous ratings. An RFC is a supervised machine learning model that consists of a collection of decision tree classifiers (DTCs) that are trained on different random subsets of the data and features. An RFC makes predictions by taking the majority vote of the DTCs. A random subset of the data means that each DTC is exposed to a different sample of the original data, which reduces the correlation among the DTCs and increases the diversity of the ensemble. A random subset of the features means that each DTC considers only a subset of the available features when looking for the best split at each node, which reduces the variance and prevents overfitting. We implemented the RFC using the Python scikit-learn library.

RFC performed slightly worse than GBT, better than DTC, and about on par with Logistic Regression.

Decision Tree Classifier

We also used a decision tree classifier (DTC) to predict the recommendation of tracks for different users based on their previous ratings. A DTC is a supervised machine learning model that splits the data into smaller and smaller subsets based on a set of features and rules. A DTC makes predictions by following a path from the root node to a leaf node that corresponds to a class label (sklearn.tree.DecisionTreeClassifier, 2023). A root node is the topmost node that contains the entire data. A leaf node is a terminal node that contains a single class label. A split is a decision point that divides the data into two branches based on a feature and a threshold. The feature and the threshold are chosen to maximize the information gain or minimize the impurity at each split. We implemented the DTC using the Python scikit-learn library. DTC performed the worst out of the ML classifications we tried by about 0.02 points, which wasn't terrible compared to many (but not the best) of the naive search algorithms or matrix factorization.

Logistic Regression

We also used a logistic regression (LR) classifier to predict the recommendation of tracks for different users based on their previous ratings. An LR classifier is a supervised machine learning model that uses a logistic function to model the probability of a binary outcome based on one or more features. An LR classifier makes predictions by estimating the odds ratio of the positive class using a linear combination of the features and a set of coefficients. The coefficients are learned by minimizing the log-loss function using an optimization algorithm such as gradient descent or Newton's method. We implemented the LR classifier using the Python scikit-learn library. Logistic Regression performed slightly worse than GBT, better than DTC, and about on par with RFC.

Ensemble Method and Conclusionary Comments

The final method we used was to linearly ensemble our set of previous submissions and scores. The basic idea behind a linear ensemble is to create an optimized linear combination of all of the predictions to approximate the ground truth solution. This optimization problem reduces down to a classic least squares solution, in which the only part of the solution we don't have is the ground truth solution X.

$$\arg\min_{\mathbf{a}} \|\mathbf{x} - \mathbf{S}\mathbf{a}\|^2 \qquad \Rightarrow \qquad \mathbf{a}_{LS} = \left(\mathbf{S}^T\mathbf{S}\right)^{-1}\mathbf{S}^T\mathbf{x}$$

Through some clever math we can rewrite the S transposed X part of this equation as a vector of N(2Pi - 1) values, where Pi is the corresponding score of each kaggle submission. Once we have aLS, we can easily create our linear combination of each submission, and choose the recommended and not recommended tracks for each user. There is one caveat to this solution, and that is duplicate solutions cannot be included in the ensemble. In our case, having duplicate submissions and scores caused the inverse of S transposed dot S to be unsolvable, and so we had to remove a couple of duplicate solutions.

Overall, the ensemble method performed the best out of all of the methods we tried, as expected. We did try and take the ensemble submissions and results and feed them back into the whole ensemble method, but quickly found this had diminishing returns.