2011 KDD Cup Challenge

Brandon Vargo

16 December, 2011

1 Getting Started

The 2011 KDD cup consisted of dataset relating to music. The object of the cup was to predict the rating of artists, genres, or songs using a set of existing user ratings as a basis for a model. Artists could have many albums, each of which had a track. In addition, albums and tracks were related to genres. All music objects were represented as integers, so that no additional additional data could be gathered about the music. Ratings were on a scale from 0-100.

I started off the project by looking at an overview of the file format and looking at the example files. In class, Professor Bakos created a database. I originally started to go this route, but there is more setup for sqlalchemy in Python than there is for ActiveRecord in Ruby. In addition, I made the observation that each object in the sample data was represented by a unique integer, even among other types of data, so it really was not necessary to have multiple tables if my algorithm did not take into account the difference between an album and an artist. My algorithm, described below, only cares about ratings, so this data is irrelevant. Thus, I did not need a large database with different types of objects. I simply needed a way to represent associations of integers. I switched to using Python dictionaries for storing the data. While this is an in-memory approach, limiting the scope of the data the program could handle, I was not concerned, and it far easier to experiment with ideas.

2 Core Idea

I postulated that the best way to predict a rating for a user A would be to find other users, B and C, who are similar to user A. If B and C had rated the item that the algorithm is trying to predict, then their ratings can be combined in some way to produce a prediction for A's rating. I took the simple approach of a weighted average, based on the similarity value and support, of the ratings of the K nearest users. Support, in this case, is defined as the number of ratings that two users share in common. The similarity score I used was the sum of the squares in the differences of ratings for all ratings shared. By adjusting K and by having a large dataset, I hoped that the data would yield a correct solution, rather than trying to create a complicated algorithm to perform the rating predictions. In fact, as you might have noted, this algorithm does not take genres or artists into account as separate entities at all. Instead, it only cares about how many rated items are shared between two users and the actual rating values themselves.

3 Implementation

I implemented the above ideas using Python. As mentioned in the first section, I used Python dictionaries as my core data structure, rather than a full database, for ease of exploration of the data. The program follows the following steps in order to produce the predictions:

1. The data is loaded from the training and validation files. Album and track data is also loaded, but it is not used.
2. Each user is compared to every other user for similarity. The final result for each user is sorted by the similarity sore and then the support count. In order to ensure that small support counts do not influence the prediction results, a minimum support is required for inclusion in this list.
3. When an item needs to be predicted for a given user, the first K nearest users who have also rated the song are found using the above sorted list of similar users. The ratings of these users are combined in a weighted fashion based on the similarity and support in order to produce the final rating.

4 Results

In order to analyze the results, I predicted each rating in the validation set and then compared to the validation value. I stored the differences and calculated the average error, in rating points.

The algorithm works surprisingly well despite its simplicity. On the sample dataset, with some tuning, I got the following results using the 5 nearest neighbors and a minimum support count of 5:

- On ratings where there were enough similar users who had rated the item, the average rating error was 6.56 out of 100.
- Across all ratings, the average rating error was 23.06 out of 100.

One major drawback of my solution is that it requires a lot of data in order to be accurate. Given the small test dataset, there simply was not enough data in order to cover all of the predictions, leading to the large errors. If a rating could not be predicted, an average value of 50 was returned by the algorithm. As the results show, if these are excluded, then the accuracy of the algorithm is quite good (less than half of a star on a 5 point scale). This indicates that, had the algorithm been run on the full dataset, then the results would have been much better on average.

In order to test this theory, I downloaded the full dataset. As I do not have the computational resources to run my algorithm on the full dataset in a reasonable amount of time, even if I had removed the in-memory limits of my program, I created a large subsample consisting of approximately 100x as many training ratings. Even with this larger dataset, there were still ratings that could not be predicted due to the lack of data. Even so, the average error decreased to 17.36. This indicates that "throwing" more data at the problem did lead to better results, and it is likely that running the algorithm on the full dataset, which is itself almost 23000x as big as the original sample dataset, would produce far better results than I am seeing with the smaller datasets.

On an interesting note, I compared my base results with the results returned after rounding the rating to the nearest integer divisible by 10, since I noticed that the sample dataset had many ratings that were divisible by 10. The results improved. By rounding, I decrease the error in the cases where the algorithm was within 5 points of the actual rating. This decreased the overall error by about 0.5 with my medium-sized dataset. This is not a large decrease, but I was surprised that the rounding resulted in a decrease at all.