Automatically Generating Musical Playlists

CS 229 - Final Project Report

Paul Julius Martinez – Calvin Crew Studebaker (From CS 221)

December 12, 2014

1 Introduction

Grouping similar songs together is a problem that has received much attention in the music industry. Algorithms that classify songs are an integral part of products like Pandora and Apple iTunes Genius. Creating playlists of songs is a complicated process, since a playlist depends heavily on the musical taste of the playlist creator. It is not immediately clear, even to a human, what properties unify the songs in a given playlist. Our project attempts to partition a set of songs into the playlists that a human would, based on simple popularity and audio features of each song.

2 Problem Definition

The problem as stated above is notably vague. To narrow down the problem space into one of tractable scope, we present the following simplifications, with justification, an objective error function that we will use to judge our results.

2.1 Problem Simplifications

One issue regarding playlist generation is the question of how many playlists to create. In a carefully curated music library, a listener may contain dozens of playlists. While certain methods (such as K-Means clustering) may be well suited for approaching such a problem, other methods we would like to use will work better if we assume that we are simply trying to partition a set of songs into two distinct playlists. Additionally, for simplicity, we will assume the two playlists are disjoint (that no song appears in both playlists).

In cases where it may be relevant we may also choose to assume that the playlists we are creating are of equal size. This is relevant when choosing our error function, explained below. We will also choose to ignore any order in a playlist.

2.2 Error Function

To be more formal, we can state our problem as follows: Given a training set T of N sets S_i of songs, and "correct" partitions P_1^i, P_2^i, create a function F from a set of songs to two sets of songs, so as to minimize the following error function:

$$\varepsilon(F) = \frac{1}{N}\sum_{i=1}^{N} L(P_1^i, P_2^i, F(S_i))$$

where L is a loss function defined for a single training example. We want L to compute the difference between the generated partition and the human-labeled "correct" partition. We can do this by simply counting the number of songs that are in the wrong playlist, then to normalize we will divide by the total number of songs. For two given playlists P_1 and P_2 and two generated playlists A and B, we can express this as $|P_1 - A| + |P_2 - B|$. Since we assume that we are partitioning into two equal sized playlists, these terms will be the same, by symmetry, so we can just take one and divide by the number of songs in single playlist. More importantly however, since the playlists have no inherent order, we must consider each possible match up between the generated playlists and the given ones, that is both, $|P_1 - A|$ and $|P_1 - B|$. Thus our error function for a single training example is:

$$L(P_1, P_2, (A, B)) = \frac{\min(|P_1 - A|, |P_1 - B|)}{|P_1|}$$

As an example, consider a partition of the numbers 1 through 8, with a correct labeled partition of P_1 = 1-4 and P_2 = 5-8. Then if we have $(A, B) = (1, 2, 4, 5, 3, 6, 7, 8)$, then $|P_1 - A| = |3| = 1$, while $|P_1 - B| = |1, 2, 4| = 3$, so $L(P_1, P_2, (A, B)) = 1/4$, which can roughly be interpreted as saying, "We got one out of the four songs wrong." Important to note though is that since we take the minimum of the two possible pairings, it is impossible not to do better than 50%.

2.3 Baseline

As a baseline for comparing our results, we would like to know the worst case scenario for our error function. Creating random partitions and testing their error indicated that the expected error of a random partition function is around 0.45. As a lower bound, we could have humans re-partition existing playlists, but this seemed like overly intensive process that we decided was an avenue not worth pursuing.

3 Data and Features

We used handcrafted playlists that we retrieved from SoundCloud using their public API[1]. We collected 18 playlists, totalling over 250 unique songs from which we collected both SoundCloud social data and audio feature data using YAAFE (Yet Another Audio Feature Extractor)[2]. With our 18 playlists, we generated $18 * 17/2 = 153$ pairs of playlists, containing an average of 20 songs per pair of playlists. These 153 pairs were then split into 80 training examples, two sets of 20 validation examples, used for parameter tuning and additional testing, and then a set of 30 test examples.

We used the following 13 features:

- SoundCloud Playback Count
- SoundCloud Download Count
- SoundCloud Favorite Count
- Duration (in seconds)
- YAAFE provides a feature called Loudness, indicating the energy over small time frames (1/10 of a second). Using this we generated:
  - Average volume
  - Beginning volume (Average of volume over first 10 seconds)
  - Ending volume (Average of volume over last 10 seconds)
  - Max volume (The 90th percentile of volume output)
  - Min volume (The 10th percentile of volume output)
  - Volume variance (Variance of the Loudness feature)
- YAAFE also presented a feature called Spectral Rolloff, described as the frequency so that 99% of the energy is contained below. We interpreted this as the maximum pitch at a given point in the song, and from this we generated:
  - High frequency (The 90th percentile of Spectral Rolloff)
  - Mid frequency (The 50th percentile of Spectral Rolloff)
  - Low frequency (The 10th percentile of Spectral Rolloff)

4 Approaches

4.1 K-Means Clustering with Weighted Norm

One of our first approaches toward solving the problem was using K-Means clustering to divide the set of songs into two groups. To try to improve performance over regular K-Means, we made a few changes. The traditional K-Means algorithm works using the a traditional Euclidean norm, but we instead used a weighted norm $\| \cdot \|_{\mathbf{w}}$. If $x, \in \mathbb{R}^n$, then we define:

$$\|x\|_{\mathbf{w}} = \sqrt{\sum_{i=1}^{n} w_i x_i^2}$$

Then, to learn the optimal weight $\mathbf{w}$, we used a hill-climbing algorithm that would start with a given weight vector then explore locally for weights that achieved a lower test error. To help encourage more exploration, we used a sort of beam search approach which would maintain a number of 'best options' at once.

Unfortunately, we were unable to achieve satisfactory results. Our best results had **0.30 training error** and **0.36 test error**.

4.2 Classifiers

Another approach we explored is training classifiers on our data, but there are a number of subtleties here. One question is how we use a classifier to then partition our playlist. The idea is that the classifier is useful not because it outputs an absolute score 'yes' or 'no', but because it can output a score that then be used to obtain a relative rank of songs. We can generate a partition by calculating a score for each song in a set from the classier, then sorting the songs according to the score and then splitting the list in two.

The other issue is how is that using a classifier is a form of supervised learning, but our training examples have no designated "correct" playlists or "incorrect" playlists. We considered the following approach to the problem. For each pair of playlists in our set of training examples, we can create a number of training examples for our classifier by taking each song in the playlist then assigning it a score of 1 if it's in one playlist and a score of -1 if it's in the other. Additionally, we give each song from that pair of playlists a constant feature that indicates it came from that training example. What this achieves is that we are actually are creating multiple decision boundaries, one for each training example. Each decision boundary uses the same weight vector, but different intercept terms, so that overall each song is being scored along the same dimension.

The question remains as to which playlists should have a score of 1 and which should have a score of -1. This is easier to visualize in Figure 1 below. In the top left we have our two training examples, where the O's denote one playlist and the X's denote the other. If we were to create separate linear classifiers for each, we would obtain the two classifiers in the top right corner. Now, if we throw all the examples together, with both sets of O's being the 'positive examples', then we can obtain a classifier very close to that, as seen in the bottom right corner. Both classifiers have the same slope, but the boundary is adjusted so that both correctly classify the data. A problem could arise if in one playlist the O's are the positive examples and in the other the X's are the positive examples. But then we'd be asking our classifier to classify data where the positives are in the middle and the negatives are on the outside, so the classifier will fail to obtain a low training error. So, to get around this problem, we try many possible "assignments" of the training examples as is feasible, then we pick the one that gives us the lower training error. Since with N training examples there are 2^N possible assignments which is an impossibly high number to check, we randomly tried 300, then picked the best one.

We tried this approach with two classifiers, a simple traditional linear classifier and a Support Vector Machine with a Gaussian Kernel.

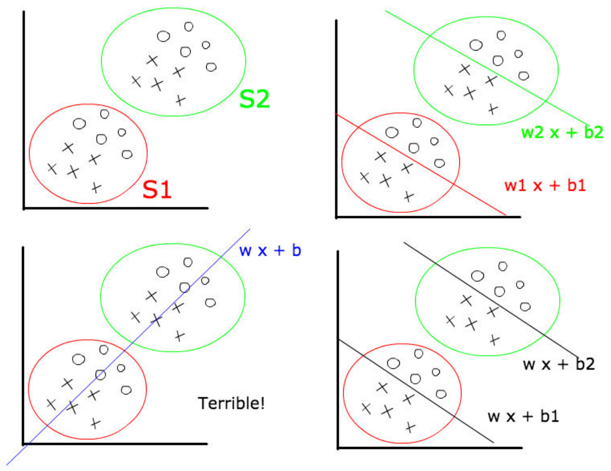

Figure 1: The effect of different assignments on classifier performance.

4.2.1 Linear Classifier

With the linear classifier we expressed all of our training examples as matrix A and their scores as a vector y and tried to find a weight vector x such that $Ax = y$. Since this is not possible in general used to the least squares solution:

$$\text{To minimize } \|Ax - y\|^2 \quad \text{set} \quad x = (A^T A)^{-1} A^T y$$

This approach worked fairly well. On our 300 random assignments, we achieved a random **training error** of **0.30**, but our **best training error** was **0.26**. Using this classifier, we achieved a **test error** of **0.28**.

4.2.2 Support Vector Machine

A Support Vector Machine, in the simplest formulation, attempts to draw a decision boundary that maximizes the minimum distance from a training example to the decision boundary. Using various mathematical tricks (the Kernel trick) we can achieve non-linear decision boundaries, but ultimately our usage of SVMs relied on the implementation in the scikit Python library[3]. Using SVMs our best classifier achieved **0.19 training error** and **0.23 test error**.

4.2.3 Classifier Analysis

We wanted to validate our intuition regarding the idea that certain assignments of our training examples led to better or poorer classifiers. We used our validation set of examples for this, and we graphed the relationship between error on the training set and error on a validation set, expecting a positive correlation.

The results show a tentative relationship between training error and validation error, which is good, but a stronger correlation would have been more reassuring. There is a however a notable difference between the performance between the two methods, and between the classifiers and the performance of the weighted K-Means, which is good.

4.3 Naive Bayes

We alternatively tried to build a Naive Bayes classifier, but ultimately this approach failed due to how we structured our training data. If we let $p(s_1, s_2)$ denote the probability that songs s_1 and s_2 are in the same playlist, then we get compute the probability of a partion P_1, P_2 as follows:

$$p(P_1, P_2) = \prod_{s,t \in P_1} p(s,t) \prod_{s,t \in P_2} p(s,t) \prod_{s \in P_1, t \in P_2} (1 - p(s,t))$$

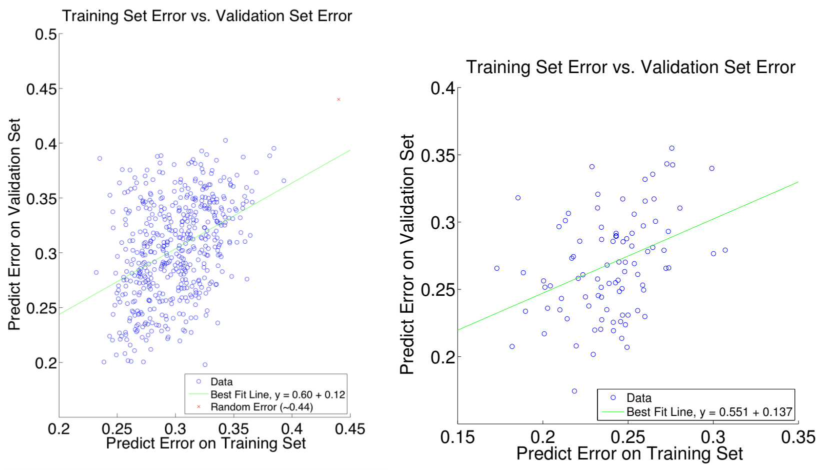

Figure 2: Relationship between training error and validation error for linear classifier (left, 300 assignments) and Support Vector Machine (right, 100 assignments)

We tried to use the training data to learn these probabilities and the compute the the partition of the playlists that had the highest probability, but we ran into a few issues. The number of possible partitions grew exponentially, so we had to resort to using a greedy algorithm for finding a high proabability partition. But the bigger problem was that our training set was small, and, crucially, our testing set used playlists that also appeared in our training set, so if we were testing on a playlist we'd seen before, it would automatically choose that as the most probable. Our Naive Bayes achieved 0 test error, but only because it was essentially getting tested on the training set.

We considered adjusting how our we trained our Naive Bayes classifier, but ultimately decided that in order for it to be truly effective, we would need a training set orders of magnitude larger than the one we already had. Given enough resources and time, however, the approach does seem promising.

5 Conclusions and Future Work

Each of our approaches performed well on both the training and testing data. With relatively simple features extracted from each song, both weighted K-means and Classification algorithms were able to learn to partition songs with impressively low training and testing error. Our SVM classifier had the best performance, achieving 0.19 training error and 0.23 test error. In our future research, we want to implement more advanced audio feature extraction such as average beats per minute, and variation in beats per minute. We would also like to examine the effects of feature templating, as well as implementing Naive Bayes with a more diverse testing dataset. We are very pleased with the results produced from the algorithms implemented in this project.

References

[1] SoundCloud, `https://developers.soundcloud.com/docs/api/reference`

[2] YAAFE (Yet Another Audio Feature Extractor), `http://yaafe.sourceforge.net/index.html`

[3] scikit Python Library, `http://scikit-learn.org/stable/`