Data Wrangling for Pace Compatibility Project

Andy Harless

The data issues for this project concerned two things:

1. Getting audio files into a form that could be used to address the question of concern
2. Getting (human-generated) label data into a form convenient to train a model

The original "raw" input files came in MP3 format (from my own music collection). Much of the processing was done using the LibROSA Python library, which initially converts an MP3 file directly to a simple time series represented by 1-dimensional NumPy floating-point array. (By default—which I accepted—the stereo channels, if they exist, are mixed to mono and sampled at a rate of 22050 Hz, which is less precise than the typical 44100 Hz used for WAV files.)

An immediate issue was that many songs are "dirty" from a tempo point of view, in the sense that the tempo is not consistent throughout the song. Particularly, at the beginning or end of many songs, there is an acceleration or deceleration, or a special section at a different tempo than most of the song. I dealt with this issue by simply removing the first 512K elements and the last 512K elements of each array (corresponding to about 24 seconds of music each: 512K/22050). This isn't a perfect solution, since it loses a lot of data, and in some cases the remaining data may still be "dirty," but I wanted to follow a simple and consistent procedure. (Songs that contained noteworthy shifts in tempo in the middle of the song, I simply discarded from the training data.)

The next issue was how to represent sound data for analysis. To keep the project simple, I chose a single representation, the Mel spectrogram, which is a common way of representing audio data to emphasize human-relevant features. By default, LibROSA calculates a spectrogram of 128 frequencies at hops of 512. For greater precision I chose a hop length of 256 instead, meaning that there are about 86 (22050/256) spectral samples per second, or about 5200 per minute. This rate allows for 16 spectral samples per beat at my (arbitrarily chosen) maximum tempo of 320 beats per minute, thus permitting me to extract a reasonable amount of information to be associated with each beat for any tempo in the range I was considering.

Before proceeding with further wrangling of the audio data, I had to make a decision about how to deal with the label data. In the form that I initially recorded my labels, a song can have up to 3 "compatible paces," but some songs have only 1 or 2. Initially I had thought I would use a whole song as the unit of analysis for machine learning and come up with some loss function to describe how well the model's outputs approximated the set of labels for a given song. But there were two problems with that approach. First, the required loss function would be quite complicated and arbitrary. Second, using a whole song as the unit of analysis would severely limit the amount of training data available. So I settled on a different approach.

Instead of using a song as the unit of analysis, I use a random clip from a song. And instead of creating a model that would estimate a set of tempos (paces) directly, I generate a set of “candidate tempos” and classify each as “wrong” or “right” depending on whether it is close to one of the human (“ground truth”) labels or not.

I generated candidate tempos as the peaks of the timewise mean of LibROSA’s tempogram. For each song, I divided it into 24-second sections and generated a set of candidate tempos for each section; then I took the union of those sets and associated that whole set with the song, as a set of candidate tempos. To generate a training (or validation) case, I choose a random song, choose a random clip from the song, and choose a random candidate tempo from the song’s associated set. Then I classify the case as either positive or negative according whether the particular candidate tempo is within 5% of one of the “ground truth” labels.

The next issue was how long to make each random clip. One possibility would have been to make all clips the same length in time, but that would mean that some clips would contain many beats at the candidate tempo and some would contain only a few. Instead I decided to make all clips the same length in terms of hypothetical beats. By “hypothetical beats” I mean beats that would occur at the candidate tempo. I chose each clip to be a length of 16 hypothetical beats. So, for example, if a song has both 80 and 160 beats per minute as candidate tempos, a clip associated with the 80 candidate tempo would last one fifth (16/80) of a minute (i.e., 12 seconds), and a clip associated with the 160 candidate tempo would last one tenth (16/160) of a minute (i.e., 6 seconds).

Since the clips (as decided above) had different lengths in time, but the spectral data were all sampled at the same time rate, I had to resample the data to put them in a consistent form. (An alternative would have been to use features that could be generated in a consistent way from different time lengths, but I chose not to go that route.) Resampling turned out to be the hardest part to code. I used Pandas time series methods to upsample and then downsample the data. The result always has 16 samples per (hypothetical) beat, but the initial upsample goes to the least common multiple of 16 and the original number of samples per beat, so as to avoid unnecessarily losing information in the resampling process. (It turns out that Pandas time series methods are not a very efficient way of doing this, but at least they work.)

After resampling, the “X” data for each case consist of a 256x128 floating point array where one dimension represents time and the other represents audio frequency (pitch). The next wrangling step was to reshape the array to 16x16x128. One dimension then represents the immediate progression of time over the time slice associated with one hypothetical beat, and the other dimension represents the time between successive hypothetical beats. For positive cases, we should expect to see a repetitive pattern across this second dimension, as each actual beat will be similar to the previous beat. For negative cases, patterns along this dimension should seem random, as they do not represent actual beats.

Finally there was the question of how to generate mini-batches of training (and validation) cases. My original idea was to generate batches on demand by sampling random clips during the training process. But the resampling code ran very slowly, which made the training process very slow. This was inconvenient, since I wanted to watch the training process in real time. So instead of generating training data on demand, I generated a huge file of training cases (which I call a "megabatch" since it reused the code originally to designed to create small batches on demand) overnight and sampled cases randomly from that file during training. (There was also a separate file for validation, generated from a separate set of songs.)

Specifically, I generated 16,384 training cases (and 4,096 validation cases). Since the training cases were generated from a single set of about 100 songs, there was not as much variety as the number 16,384 might seem to suggest. However, since the clips were sampled randomly, the phase relative to the beat was different in different clips from the same song. So there was still a lot for a machine learning model to learn.

Currently, code for the data wrangling steps is contained in the following files in the repo:

`SongsSliceDiceSetsWork2.ipynb`:
code to process each song into spectrogram and candidate tempos

`get_training_data.py`:
code to sample clips from songs and associate each clip with a candidate tempo

`SaveMegabatch.ipynb`:
code to standardize clips for use as training/validation data and produce large files thereof

`pace_params.py`:
some of the parameters used in the above files

As of now (2019-02-16), the internal documentation of these files still needs considerable improvement, and much of the code itself could do well with some cleaning up.