Brennan Coslett
Reptile Tornadoes

Framework.py

I tried to port a lot of the necessary functionality of the JavaFramework that was provided to us into python for this project and this file contains the ported functions as well as some new things I wrote. There are comments for all of the functions in the file with descriptions, so I won't go into many of them. We ended up using "STFT" for all of our stft needs as we couldn't get madmom to install and the functionality looked very similar between the two. The most important part of framework.py is calcFrameEnergies.

calcFrameEnergies(file_path, weightForHFC)

It begins by converting an imported file from .wav to stft. If weighting the bins for *High Frequency Content* (**weightForHFC = true**) it creates a numpy array the length of the num of stft frames and will sum all of the energies in all bins for each frame. The formatting on this looks weird because the stft returns a numpy array of shape (**bins x frames**). We take the square root of the abs value of each bin and weight it by (**j/numBins**) where **j** is the current bin number. I was testing with this preprocessing and taking the sqrt of our frame energies improved our final F-measures by around 5%. If using *SpectralDifference* then the difference is calculated for each bin in each frame between the current frame and the previous and then the differences are summed. Finally, the values are normalized to a maximum value of 1 to allow for easier analysis.

MusicAnalzyer

The analysis has been wrapped into the musicAnalzyer class. On initialization it takes a given input directory, any subset of the files in said directory, whether to use *HFC* or *SD* for onset detection, and whether to plot the values returned by **calcFrameEnergies** (mostly for debugging). The evaluation functions were used during development to calculate the *F-Measure* for our different approaches.

detectOnsets()

This function will iterate through all wav files in **self.input_dir** from the subrange specified during initialization. For each file it will **calcFrameEnergies** and if plotting is enabled plot the graph for that *.wav* file. It creates a list **filePeaks** to store all peaks that are high enough. The peak picking algorithm iterates over the frames by batches, size **self.batch_size**, and calculates the minimum value to consider a peak at the mean value of

frame_energies between [*value – batch_size : value + batch_size]*. This moving average creates a much more accurate function than using a predefined value to increase by. We found that the most accurate peak picking for our algorithm meant setting the minimum at the **average * 1.08.** The function then creates the ***.pr** *file and writes the time value of each onset (peak in filePeaks,* ***a frame index****)* * frame_length.

extractTempo()

Our tempo extraction method is fairly simple. The function reads the list created by the onset detection function and turns it into an array. It computes the inter-onset-interval between each onset **(onset[i] – onset[i-1])** if the IOI is in between .25 (*240 BPM*) and 1 (*60 BPM*) we will count it towards the total list of “useful” onsets. I found that this range gave the best accuracy when working a larger range of files as when we used a strict .3-1 lag interval (60-200 BPM) we were not logging any valid onsets for some songs. Conveniently the onsets that fell in the 200-240 range were usually meant an octave error. We calculated the mean() of the list of all valid onsets for each song and used that as our lag estimate for the song. Therefore, to get the tempo we took **60/avgIOI.** If the tempo we got back was higher than 200BPM we divided by 2 and assumed an octave error. This approach seemed to give us a fairly accurate tempo extraction without needing autocorrelation.

detectBeats()

The beat detector builds upon the previous two functions. It will import the list of onsets and the tempo for each file. We used librosa to get the total length of the .wav file in question. Then we will iterate through the song in **chunks equal to the one lag period.** At each of these points we go through the list of onsets and check if any of the onsets are within 10% of a lag period and if they are then we will add them to our beat array. While making sure that they’re not added twice. Then we write the beat list to a prediction file.