Key Constraint

Francis, Kelsey *Martin, Christopher*

Abstract

The Key Constraint project aims to analyze a piece of recorded music to determine its dominant key and utilize this information to produce a simple electronic instrument to accompany the recording, playing only notes from same key of the original song.

Source code

The code for this project is hosted on GitHub:

https://github.com/kelseyfrancis/keyconstraint

Key Identification

Related Work

Many techniques for identifying the tonality, or key, of recorded music have been previously investigated. Fujishima introduced *pitch class profiles* [1], a longstanding approach that has been built upon by many, including Temperly [2]. Our algorithm hinges on pitch class profiles.

Many other techniques have been proposed that we did not utilize but researched. Rizo et al. proposed a *tree model* approach [3] that is limited to Western music, i.e., 12 semitones per octave. E. Chew presented a *spiral array model* that can also detect changes in key [4], also restricted to music that exhibits tonic triad dominance, typical of music in the Western tradition. Sheh and Ellis employ EM-Trained Hidden Markov Models [5]. These few works are just a small sample of the investigation into this topic.

Gomez and Herrera presented a technique [6] similar to the approach we took: extract features from the audio in a form similar to the pitch class profile, train a learner with those features based on some model, and use the trained learner to classify features extracted from the key-unknown audio signal.

I/O and Sample Representation

Our approach to identifying key hinges on first extracting musically meaningful features from the audio signal in question through spectrum analysis. The first step to analyzing an audio signal is, of course, to read an audio file into memory. We did not use a library for this task, because we wanted to learn the WAVE format. We read in a WAVE files into an array of double-precision floating point samples. We then downmix from stereo to mono, if we detect that the file is indeed multi-channel. Again, we wrote the downmixing code ourselves to learn how it works.

Much of our music library consists of CDs ripped to a single, large WAVE file accompanied by a cue sheet that indicates the offset at which each song starts and its title. We again wrote the code to read these cue sheets ourselves. We altered our WAVE file I/O code such that it could directly read in only samples between certain time offsets. This was critical because such a large WAVE file could not be loaded into memory and then split. We also wrote code to write the separate tracks out to disk for convenience in playback.

Spectrum Analysis

Once the audio is in memory and downmixed, we perform a spectrum analysis on the audio by running an FFT, using the JTransforms library [7], on overlapping frames of audio to which a window function has been applied. We again implemented the window functions we used in order to learn about them in more detail; we considered Hamming, Hann, and the function used by Ogg Vorbis. We eventually settled on Hamming because it experimentally seemed to work the best.

We ensure that, given the sample rate as read in the WAVE header, the number of FFT buckets will provide a resolution of 1 Hz, which makes certain that each pitch class (discussed below) covers at least one bin, even in the very low octaves, and minimizes overlap of pitch classes between bins. A future performance improvement would to first apply a band-pass filter to only consider certain octaves, then lower the number of bins to only those needed to resolve pitches in that band. An alternative to this would be to use a Constant Q transform [8], which has logarithmically sized bins, a perfect fit for finding musical pitches, which scale logarithmically between octaves.

Pitch Class Profile

Using these FFT results, we construct a distribution of pitches within the signal to serve as our feature vector for the audio signal. This pitch class profile is comprised of how often each of the 12 equal-temperament semitone pitches typical of Western music occur within the the signal, in any octave, e.g., how often the note A occurs in any octave. Originally, we included in this vector of features another distribution that considered each octave separately, e.g., A4 vs. A5, etc, but this had poor results in the machine learning classification step, which lead us to believe these features were not helpful in determining tonality.

The pitch class for a given frequency can be determined by first assigning a real number along a linear space to each frequency: $69 + 12 \log_2 (f / 440)$, where 69 is the A above middle C [9]. This translates frequencies that increase logarithmically with increasing pitch into linearly increasing pitch values, where whole numbers represent the typical Western notes. We apply this formula for each FFT bin's centerpoint frequency, and heuristically disregard any bin not within 0.1 of a whole number (we again experimented with several values here). We then round this value and use it as an index, mod 12, into our distribution of pitches, where we keep a running sum of the magnitudes of each corresponding frequency. Finally, we normalize the resulting vector such that the prevalent pitch has a value of 1.0, so that songs of different loudness can be compared.

Machine Learning

The next key identification step is to extract the pitch class profile of many audio files of which we know the musical key. For this we used a complete collection of Chopin's piano works, as well as some other classical music. The program is passed a cue sheet or one or more WAVE files and in turn prompts the user for the label to apply to each track. The features are extracted as described above, and saved to an ARFF file (labels.arff, by default). The labels for the Chopin piano works are already present in labels.arff. To add labels, run:

```
./identifykey.sh -v --label -f songToLabel.wav
```

With these labeled features, we train a classifier, using the Weka library [10], which we then use to classify features extracted from a song not previously labeled. To predict the key of a song, run:

./identifykey.sh -f songToPredict.wav

The identifykey program has many other options, such as --dist for displaying the probability distribution of predicted keys, rather than just the most probable. For help, run:

./identifykey.sh -h

Please note that this program was compiled and run with Java 1.6 and built with Maven 3.0.2.

We experimented with various numbers of labeled song and with various classifier algorithms. The default is a combined classifier that uses multiple underlying classifiers and computes a weighted average of their results, with weights assigned heuristically based on which classifiers tended to do the best on their own. A particular classifier can be specified:

./identifykey.sh --classifier randomforest -f songToPredict.wav

The output of the command is simply the predicted key, unless options such as --verbose, --title, or --dist are specified, in which case it outputs more information. Upper case keys indicate major, while lower case indicate minor.

Results

We found that even with a small number of labels, this approach works quite well. In the video we submitted with this report, we show that training with a single CD of Chopin piano works, which cover most of the 24 major/minor keys, results in fairly accurate prediction of keys on songs on a CD of Beethoven piano works. The key of all but one song is either correctly identified, or the relative major/minor key is predicted, or a major/minor key either up a fifth or down a fourth on the circle of fifths is identified. Such mistakes are reasonable from a music theory standpoint and can likely be attributed to harmonics in the signal. Further, such keys, while incorrect, would still result in pitches that are mostly harmonic with the song, because the keys are adjacent on the circle of fifths. *Correction to the narration in the video:* For one song in this set, a completely incorrect key was predicted, which we mistakenly failed to mention when shooting the video.

We also labeled a large amount of Chopin piano music, after which key prediction seemed to improve somewhat over a wider set of test songs. When we added labels for a couple CDs of Corelli orchestral music, however, prediction seemed to get worse. Our hypothesis is that our approach is sensitive to music with different instruments, and probably different genres as well. A related limitation to this approach is that it is tedious and time-consuming to label large volumes of diverse music, and simply acquiring that music can be a limiting factor.

Future work

Future steps to improve on this approach include 1) using a band-pass filter on the audio signal to only include frequencies likely to be musically relevant; 2) experimenting with different Fourier transforms, window sizes and overlap amount, and window functions to find the most suitable for musical signals; and 3) conducting large-scale experiments with thousands of labeled songs and with many different combinations of heuristic parameters and classifiers.

Synthesis

music.py

We built a library for some music theory to do calculations with musical notes. Most notably, it parse note names, determines scales for keys, calculates note frequencies, and shifts notes between keys.

synth.py

This is a basic modular synthesizer. This section of the project does not utilizing any audio libraries, primarily just because we wanted the experience of developing our own synthesis software from the ground up. It supports oscillators in sine, triangle, and square waveforms and modulation of amplitude and frequency.

Synthesis modules are implemented as classes with two primary methods:

```
next(t, n) : samples
        t : time increment (float or numpy.array of floats)
        n : number of samples requested
        samples : numpy.array of floats

liveness() : [ 'live', 'sleep', 'dead' ]
```

"next" is a vector-valued function for obtaining a chunk of output samples. The parameter "t" is generally given as 1, and varies when frequency modulation is utilized. The "liveness" output is used for performance to remove components of additive modules that are not currently audible.

To sidestep some performance concerns, we also have a wavetable module type that memoizes fixed-length samples. Once the instrument's wave tables for each note are generated, their addition is sufficiently performant for playing in real time.

How to run

"play/scale.py" demonstrates how we can play a scale. This script accepts one parameter: the name of key. Capital/lower case letters denote major/minor keys. "#" and "b" denote sharps and flats. For example, to play a scale in C major followed by another in G-sharp minor:

```
python scale.py 'C' && python scale.py 'g#'
```

Please note that the audio playback requires that you have "aplay" installed, and that this has been tested only with Python 2.7.2.

The instrument can be played by itself using "play/solo.py". Its first argument is the name of the key into which notes are transformed, and the second is a string which is used to search the names of connected MIDI devices. For example, to use the virtual device vkeybd to play in D, launch it and then run:

```
python solo.py 'D' Virtual
```

The synthesis is united with the key detection results by "play/accompany.py":

```
python accompany.py Virtual ../music/aria.wav
```

This runs the detection algorithm on aria.wav, begins playing it, and enables the instrument which can play along in the same key. The playback transforms notes from the C major (the white keys on a piano) into the key of the song. and other notes are silenced.

Steps for future improvement

Wave tables should be cached on the filesystem so they don't need to be generated anew each time the program launches.
To distribute this software for popular use, we would need to add a graphical interface to select a file and pause/restart its playback.

Division of labor

Chris wrote all of the synthesis code. Kelsey wrote all of the key identification code. Chris also advised on various machine learning topics, e.g., appropriate feature vector size, etc, since he has more formal knowledge of that area. Kelsey also advised on music theory topics with respect to both identification and synthesis, since he has more formal music training.

References

[1] Fujishima, T. Realtime chord recognition of musical sound: A system using Common Lisp Music. In *Proceedings of the International Computer Music Conference*. Beijing. 1999.

[2] D. Temperley. Improving the krumhansl-schmuckler key-finding algorithm. In *22nd Annual Meeting of the Society for Music Theory*. Atlanta. 1999.

[3] D. Rizo, J.M. Inesta, and P.J. Ponce de Leon. Tree model of symbolic music for tonality guessing. In *Proceedings of the Int. Conf. on Artificial Intelligence and Applications, AIA*. Innsbruck, Austria, 2006.

[4] E. Chew. The spiral array: an algorithm for determining key boundaries. In *Proceedings of the Second International Conference on Music and Artificial Intelligence*, ICMAI. 2002.

[5] A. Sheh and D. P. Ellis. Chord segmentation and recognition using EM-trained hidden Markov models. In *Proceedings of the International Symposium on Music Information Retrieval*. Baltimore. 2003.

[6] Gomez, E., and Herrera, P. Estimating the tonality of polyphonic audio files: cognitive versus machine learning modelling strategies. In *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR04)*. Barcelona. 2004.

[7] JTransforms. http://sites.google.com/site/piotrwendykier/software/jtransforms

[8] Constant Q transform. http://en.wikipedia.org/wiki/Constant_Q_transform

[9] Pitch class. http://en.wikipedia.org/wiki/Pitch_class#Integer_notation

[10] Weka. http://www.cs.waikato.ac.nz/ml/weka/.