Physics 240 Final Project: Identifying Musical Instruments

Aaron Deich

May 19, 2015

Abstract

In this proof-of-concept project, I construct a program meant to identify the source of sound recordings of a constant note played by a musical instrument, based on a set of training data. This approach uses machine learning to classify the sounds on their normalized power spectra. Each power spectrum is normalized around its fundamental frequency, the determination of which requires automated peak-detection. Using a training set of 4 different instruments (humming, piano, acoustic guitar, and recorder) each with 10, 1-second sound recordings of different pitches, my program achieves about a 50% identification accuracy for sound samples outside of the training set and about a 90% identification accuracy for sound samples already in the training set.

1 Introduction

When you listen to a sound recording of a musical instrument playing a constant note, how is your brain able to identify what instrument is making the sound? Well, *how* your brain actually works is well beyond the scope of this paper, but by considering just the basic physics of sound, there are a few parameters we can rule out: if you adjust the volume knob, the recording becomes louder or softer but still sounds like the same instrument. Thus the *amplitude* of the signal doesn't affect the identification. Likewise, the instrument can play higher or lower notes without affecting its likeness. Thus the *fundamental frequency* doesn't much affect identification, either.

It turns out that the identifying characteristics of a constant sound are contained entirely in how much of each frequency is present in its signal, relative to the fundamental frequency: this is called the sound's *power spectrum*. For constant musical notes in particular, there are almost always spikes in frequency at the first 10 (give-or-take) integer-multiples of the fundamental frequency; these spikes are called the *overtones* or *harmonics* of the sound; their relative heights and widths vary between instruments, determining together the distinguishing *timbre* of the instrument's sound.

From a data-identification perspective, a sound's volume and fundamental frequency each have far greater weight than the sound's overtone series. Thus it must be that our brains account for both frequency and intensity before identifying a sound.

In this project, I attempt to perform the same task of audio identification but with a computer. The aforementioned accounting for sound volume and fundamental frequency are mostly what I focus on.

My program takes a few dozen sound samples from different instruments, fourier transforms the signals, and adjusts each spectrum so that the lowest harmonic peak is at a constant place. It then performs data identification with Machine Learning. Machine learning, and in particular, Support Vector Machines, are well-suited to this kind of classification [1]. A Support Vector Machine constructs the best hyperplane in a high-dimensional space that most separates the different classifications of data. [2].

2 Methods

This section describes the components of my program.

2.1 Data Collection

I recorded myself playing piano and guitar, recorder, and myself humming. I used my Macbook Air's built-in microphone, making sure that the computer's fan wasn't running during recording.

I ran into problems using standard music-recording software (Garageband and Logic Pro) because they apparently add metadata to the WAV files which scipy's reader is unable to parse. I ended up using the free software Audacity for recording the sounds. After making a recording in Audacity, I select a region of that recording and export to a wav file. For the sake of lower computing volume, I've chosen a sample rate of 16,000 Hz.

2.2 FFT Implementation

In order to read a sound file and obtain an array of signal values, I use `scipy.io.wavfile.read()`. I perform a Fast Fourier Transform on this data with `numpy.fft.fft()`. My wrapper function around the FFT returns only the absolute value of the complex transform and only its values for positive frequencies.

2.3 Peak Detection

As mentioned above, for the purposes of comparing instruments, I first need to "normalize" all recordings such that their fundamental frequencies are all shifted to some agreed-upon constant. This requires that I algorithmically detect a sound's fundamental frequency with good precision. Since a sound's fundamental frequency occurs, by definition, at the lowest-frequency peak in the sound's power spectrum, my task becomes one of peak-finding.

My implementation is based off the Matlab program of `http://terpconnect.umd.edu/~toh/spectrum/PeakFindingandMeasurement.htm`. This looks for maxima in the data by smoothing the power spectrum and looking at where its 1st-derivative is zero. Specifically, here is the general algorithm I'm currently using in `findpeaks.py`:

1. Given the power spectrum data, compute its 1st- and 2nd-derivatives, dy/dx and d^2y/dx^2. I use the simple Centered Derivative, $\mathrm{d}(a_n) = (a_{n+1} - a_{n-1})/2$. To account for the derivative being undefined for both end points, I don't measure the derivative there, instead setting the derivative at the endpoints to 0 (it's nice to keep all arrays of the same dimension).
2. Smooth these derivatives with a boxcar averaging. This just means that for a given array to be smoothed, starting at the nth item, we iterate through the array, replacing each element with the average of its n closest neighbors. You can

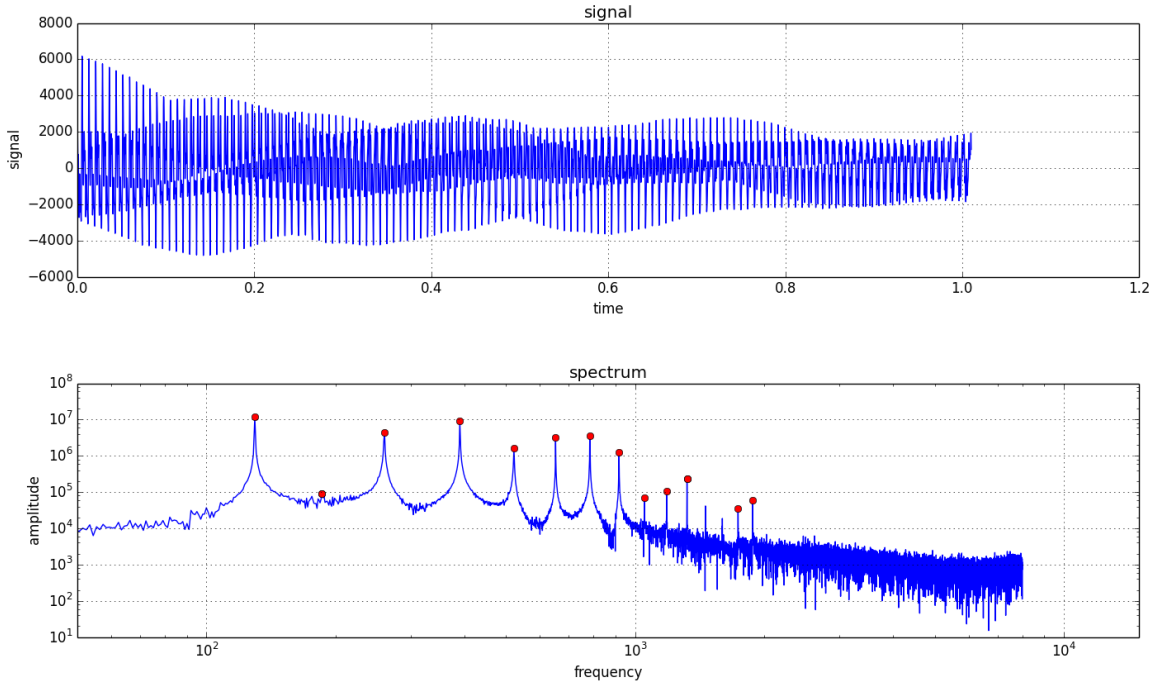

Figure 1: Piano, playing a C3 (130Hz) for 1 second. This recording's spectrum is shown below, with the red circles at detected peaks. Notice one false positive peak sitting in between the first two real peaks. Preventing false positives is difficult and requires tuning of the smoothing parameters.

see this smoothness in Figures 2 and 3, showing the 1st and 2nd derivatives of the piano sound sample of Figure 1.

3. Now we check for likely peaks: For iteration index j in (0, len(y)-1): Record all indices j where
   (a) The 1st-derivative array element, d1[j], passes through 0. i.e. d1[j] $\cdot$ d1[$j+1$] < 0 or d1[$j-1$] $\cdot$ d1[j] < 0.
   (b) The 2nd-derivative array element, d2[j], is larger than a specified threshold. This acts as a filter for mini, in-between peaks we don't care about.
   (c) We fit a large degree polynomial (30) to the data. Any peak below this fit is ruled out.
4. The locations of derivative-zeros do a good job of locating unique peaks, but they do a poor job of hitting the precise *top* of the peak. So for each peak guess, now we simply take the maximum element in a small local region around the peak guess. When the initial peak guess is fairly good, this hits the peak precisely. The only concern here is not to choose the max-element region too large or else we end up locking on to an adjacent, larger peak.

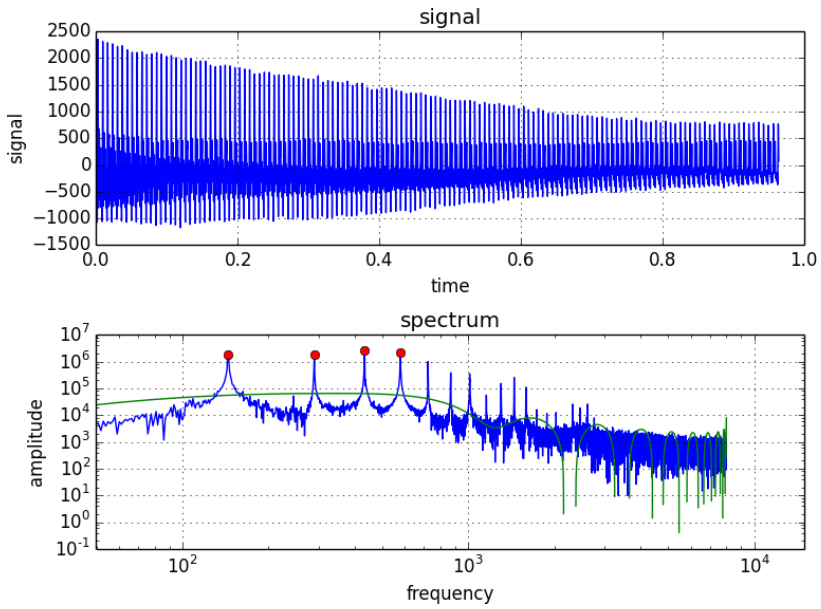

Figure 2: A guitar's signal and spectrum. The green line is a 30th-degree polynomial fit of the spectrum data; all peak candidates below the line are discarded.

2.4 Spectrum Normalization

Classification schemes require that their input data be as consistent as possible in parameters which are irrelevant to classification (such as length of input arrays). Therefore I try to make my input data as similarly-formatted as I can. This formatting consists of the following steps:

1. The first peak should occur at a fixed index j, in frequency. I chose j=100, so as to include the a decent part of the spectrum leading up to the peak. To readjust this spectrum, I `numpy` 'roll' it left or right. If I roll it right, I pad the beginning of the array with zeros.
2. The first peak should occur at a fixed height; I've chosen $h = 1$. To get this, I divide the whole FFT array by the height of the first peak.
3. The array is of fixed length. Given that my FFT data arrays are usually around length 8000, I chose a fixed length of 5000. I chop off the end of arrays that are longer than 5000 and discard arrays which are originally shorter than 5000.

2.5 Classification with Machine Learning

For identifying normalized spectrum vectors, I use a Python implementation of Support Vector Machines (SVMs) called `scikit-learn`. The classification function requires that I supply it with a list of data arrays (each representing a normalized spectrum) and it requires for each array a string identification (e.g. 'piano', 'guitar', etc.).

Training the SVM takes only about 5 seconds with my current data, so that is not a limiting factor.

I pass a given normalized spectrum to the trained SVM object and it prints out the string description of its choosing.

3 Results

This type of machine learning (classification with SVMs) sadly gives no additional data beyond a best guess for each identification. Thus my only way to test its accuracy is to count the number of sound samples the SVM correctly identifies.

For sound samples already in the training set, about 90% were correct. For samples outside the training set, about 50% were correct.

4 Conclusion

The machine learning seemed to work ok, but I have no insight as to why it worked. It would be interesting to record more data and see if the SVM becomes more precise.

The peak-detection worked fairly well, but could use a lot of improvement, primarily in automated smoothing adjustment. If all 10-ish harmonics of a sound recording could be precisely located and described, it would be very interesting to characterize a sound with these peaks alone.

References

[1] Joachims, T. 1998, Springer. `https://eldorado.tu-dortmund.de/bitstream/2003/2595/1/report23_ps.pdf`

[2] Ting-Fan, W., et al. 2004, Journal of Machine Learning Research. `http://www.csie.ntu.edu.tw/~cjlin/papers/svmprob/svmprob.pdf`