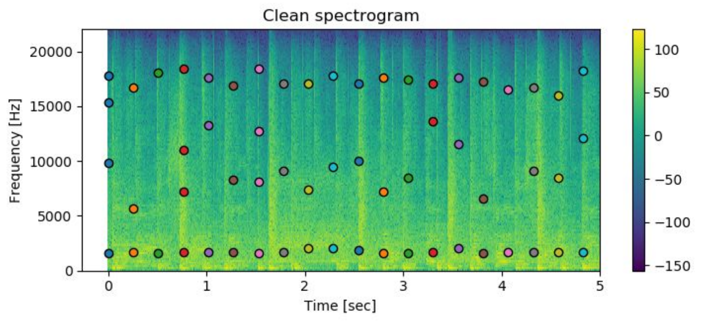
ECE 429 Final Project Report
Acoustic Fingerprinting with Wavelets

Author: Chris Druta



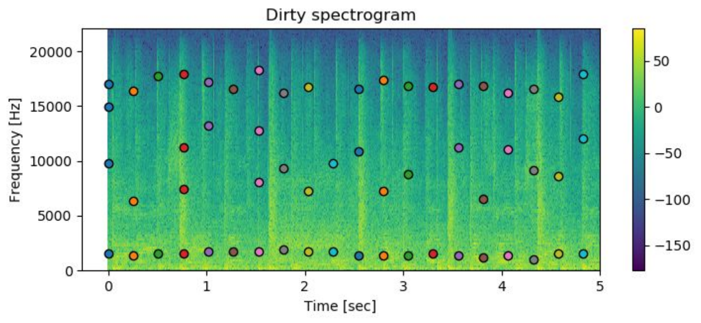


Figure 1: Fingerprint example

Abstract

Acoustic fingerprinting is the action of encoding a recorded sample of audio into hashes and detecting matches based on this generated hash. The key idea in fingerprinting audio is to hash data that will be persistent even with noise, more specifically the frequency peaks through time.

I have implemented my own peak finding algorithm for spectrograms, which contains the information that will be hashed. It can successfully identify clips when adding noise to clips digitally, and more practically when recordings clips of songs my smartphone with natural background noise.

Introduction

The fingerprinting process is quite simple and straightforward at high level.

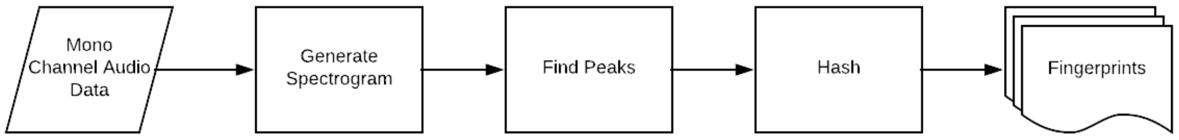


Figure 2. Fingerprinting System Diagram

Every sample of audio is fingerprinted so it can be analyzed. Our software model contains a list of songs representing a library of song data containing their full hashes to compare to. So after everything has been fingerprinted, to find matches we must search for matching hashes, and then count up and verify the match offsets.

I implemented the peak finding algorithm using the Continuous Wavelet Transform, which will be discussed more so in the next section.

Algorithm

My algorithm primarily focused on extracting information from the spectrogram of a given audio sample. The way I did this was by first finding the 'significant' time axis peaks of the spectrogram. The first thing I needed to do was cleanse the data.

The cleansing of spectrogram consisted of adding a very small epsilon to all 0 values of the spectrogram, taking the log10 of it, and multiplying it by 20 to get decibels. I then normalized it by dividing everything by the max value.

To get the significant time peaks, I went through each time index of the spectrogram, and summed all values above the chosen high pass frequency of 4000 Hz. So then I had a array of sums of frequency magnitudes above 4000 Hz for each time slice.

I then applied the continuous wavelet transform on this array with a single window of width 1 (more on that later), which gave me all the time 'spike' indices of the spectrogram.

I went through each of these time spike indices and used the continuous wavelet transform with a single window width of 10 to find the frequency peaks for each slice. The final result of my algorithm was a list of time peak indices, and list of each time peak's frequency peaks.

Continuous Wavelet Transform

From my understanding and research online, the continuous wavelet transform (CWT) can be thought of as a “smoothed out FFT”, being smoothed out by convolving scalar multiples of the “Mother Wavelet” at different widths.

My algorithm only uses single windows for the CWTs. When I was playing around with multiple window wizes of differing steps, there was a lot of variation between clean data fingerprints and noisy data fingerprints, making almost no matches come when comparing. When I tried single, small valued windows, the given peaks fit the data almost perfectly much more consistently between the two sets of test data.

The given windows above and in the source code have been found by manual twiddling. I bet my hash match rate could be much higher if I empirically found and understood the effect of multiple windows for continuous wavelets transforms instead of randomly guessing.

Software Package

This project was completed all in python, utilizing popular libraries like scipy.signal and numpy.

Github Repo Link: https://github.com/chrisdruta/python-acoustic-fingerprinter

Structure

python-acoustic-fingerprinter/

./images/	Where example images
./sounds/	Where sound clips are stored
./fingerprint.py	Contains all major algorithm functions
./main.py	Driver script that tests fingerprinting & matching
./requirements.txt	Standard python package requirements text file

User Interface

There are many commented parameters in main.py and fingerprint.py the user can change. Here is a list of them, their line number, and use. **Bold** items indicate a high impact parameter.

- main.py

sd.default.device	Output sink for sounddevice lib. Required for clip playback
clipDuration	How many seconds to clip sounds for testing
clipMultiplier	How much to multiply raw clip data (amplitude) by
noiseMultiplier	How much to multiply raw noise data (amplitude) by
recordingMultipler	How much to multiply raw recorded data (amplitude) by

- fringerprint.py

highpassWc	Frequency (in Hz) of highpass cutoff
windowTime	Array of window widths for CWT on time axis
windowFreq	Array of window widths for CWT on frequency axis (*1)
fanValue	Factor of how much to hash with neighbors (*2)

Notes

1: Want a good window size that localizes peak through noise

2: Drastically increases amount of fingerprints, and chance of collisions (wrong matches)

How to Run

Make sure you have the requirements installed for your python environment and run

```
> python main.py
```

inside the root project directory.

Hashing and Matching Algorithms

Will Drevo (in references) wrote an excellent article on the fundamentals of audio fingerprinting which included a basic hashing and matching algorithm. His code implemented a whole audio fingerprinting and detection service with a database. I implemented a localized non-database version of his hashing and matching algorithms so I could more fairly compare my peak finding algorithm to some baseline. All hashing and match finding functions are heavily based off of his.

Examples and Performance

While my system can successfully detect matches from both digitally modified and recorded audio files with noise, it is rather finicky. When finding matches, there is not that much confidence when there is a lot of noise as expected.

From over on average 80 (time peaks) * 50 (fan value) = 4000 fingerprints per 5 second clip, there are on average only 1 - 4 matches. This is with a pseudo data SNR of

$$Digital\ Data\ "SNR" = \frac{clipMultiplier}{noiseMultiplier} = \frac{0.3}{10} = 0.03$$

Those multiplier values above seem to be near the max of distiguisible “SNR”. Increasing this ratio slightly increases matches which can be seen in the following examples. Anything below that ratio, matches become extremely rare.

Before getting to the tests, ultimately the performance is not that great as it should have a lot more hashe matches. I don't think this is a result of my peak finding algorithm, but my hashing and matching algorithm implementations. If I were to fingerprint each individual peak instead of pairs, I would get a lot more matches I believe.

Tests

Example Output 1: clipMuliplier = 0.3 noiseMultiplier = 10 clipDuration = 5 recordingMultipler = 6

```
> Executing task: python main.py <

Reading In Files... Done
Generating Clips... Done
Finger Printing Samples... Done

Starting Tests

Matching clip1..
Number of matches: 2
Matched with Space Jam

Matching clip2..
Number of matches: 3
Matched with Space Jam

Matching clip3..
Number of matches: 1
Matched with Ghost Slammers

Matching clip4..
Number of matches: 1
Matched with Ghost Slammers

Matching recording1..
Number of matches: 0
Failed to match

Matching recording2..
Number of matches: 1
Matched with Space Jam
```

Example Output 2: clipMuliplier = 0.5 noiseMultiplier = 5 clipDuration = 5 recordingMultipler = 8

```
> Executing task: python main.py <

Reading In Files... Done
Generating Clips... Done
Finger Printing Samples... Done

Starting Tests

Matching clip1..
Number of matches: 3
Matched with Space Jam

Matching clip2..
Number of matches: 6
Matched with Space Jam

Matching clip3..
Number of matches: 3
Matched with Space Jam

Matching clip4..
Number of matches: 2
Matched with Ghost Slammers

Matching recording1..
Number of matches: 0
Failed to match

Matching recording2..
Number of matches: 1
Matched with Space Jam
```

Note the wrong detection of clip 23 (we expected Ghost Slammers in that test). This is actually due to a bug in aligning fingerprint matches which will be explained later.

Example Output 3: clipMuliplier = 0.8 noiseMultiplier = 5 clipDuration = 5 recordingMultipler = 10

```
> Executing task: python main.py <

Reading In Files... Done
Generating Clips... Done
Finger Printing Samples... Done

Starting Tests

Matching clip1..
Number of matches: 1
Matched with Space Jam

Matching clip2..
Number of matches: 5
Matched with Space Jam

Matching clip3..
Number of matches: 5
Matched with Ghost Slammers

Matching clip4..
Number of matches: 7
Matched with Ghost Slammers

Matching recording1..
Number of matches: 0
Failed to match

Matching recording2..
Number of matches: 0
Failed to match
```

Example Output 4: clipMuliplier = 0.5 noiseMultiplier = 9 clipDuration = 10 recordingMultipler = 6

```
> Executing task: python main.py <

Reading In Files... Done
Generating Clips... Done
Finger Printing Samples... Done

Starting Tests

Matching clip1..
Number of matches: 18
Matched with Space Jam

Matching clip2..
Number of matches: 12
Matched with Ghost Slammers

Matching clip3..
Number of matches: 17
Matched with Space Jam

Matching clip4..
Number of matches: 18
Matched with Space Jam

Matching recording1..
Number of matches: 0
Failed to match

Matching recording2..
Number of matches: 1
Matched with Space Jam
```

Note the wrong detection of clip 2 (we expected Space Jam in that test)

Shortcomings and Future Improvements

As explained earlier, I believe the hashing and matching algorithms are the biggest factors affecting performance. Most importantly as seen above, incorrect matches occur when matches aren't 'aligned' correctly (this is explained in Drevo's article). I just flat out did not implement the offset alignment algorithm correctly I think, as each song match hash pair should have a constant offsets, and not add a new offset value every time. This is the root cause of the incorrect matches because under certain cases, the last found hash song id will return instead of the true song id.

In the future, empirically deriving optimized wavelet widths would greatly improve peak findings in the algorithm I implemented. Also, hashing individual frequency peaks rather than pairs of peaks I believe would increase confidence if the match alignment was implemented correctly.

References

"Continuous Wavelet Transform." Wikipedia, Wikimedia Foundation, 15 Nov. 2018, en.wikipedia.org/wiki/Continuous_wavelet_transform.

Drevo, Will. Audio Fingerprinting with Python and Numpy, 15 Nov. 2013, willdrevo.com/fingerprinting-and-audio-recognition-with-python/.

Müller, M., & Gorsche, P. (2012). Audio Content-Based Audio Retrieval. Fundamentals of Music Processing, 3, 157-174. doi:10.4230/DFU.Vol3.11041.157