Implementing an Audio Search Algorithm

MTH 438 Project Option 4

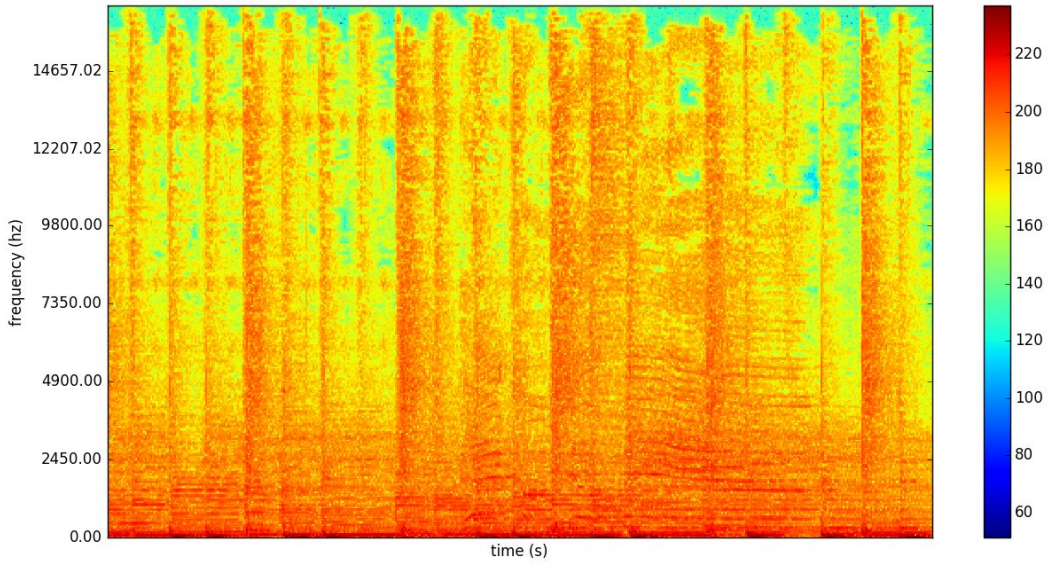
Jackson Donnelly, Nathan Margaglio, Sun Kim, 04/28/2016

Introduction:

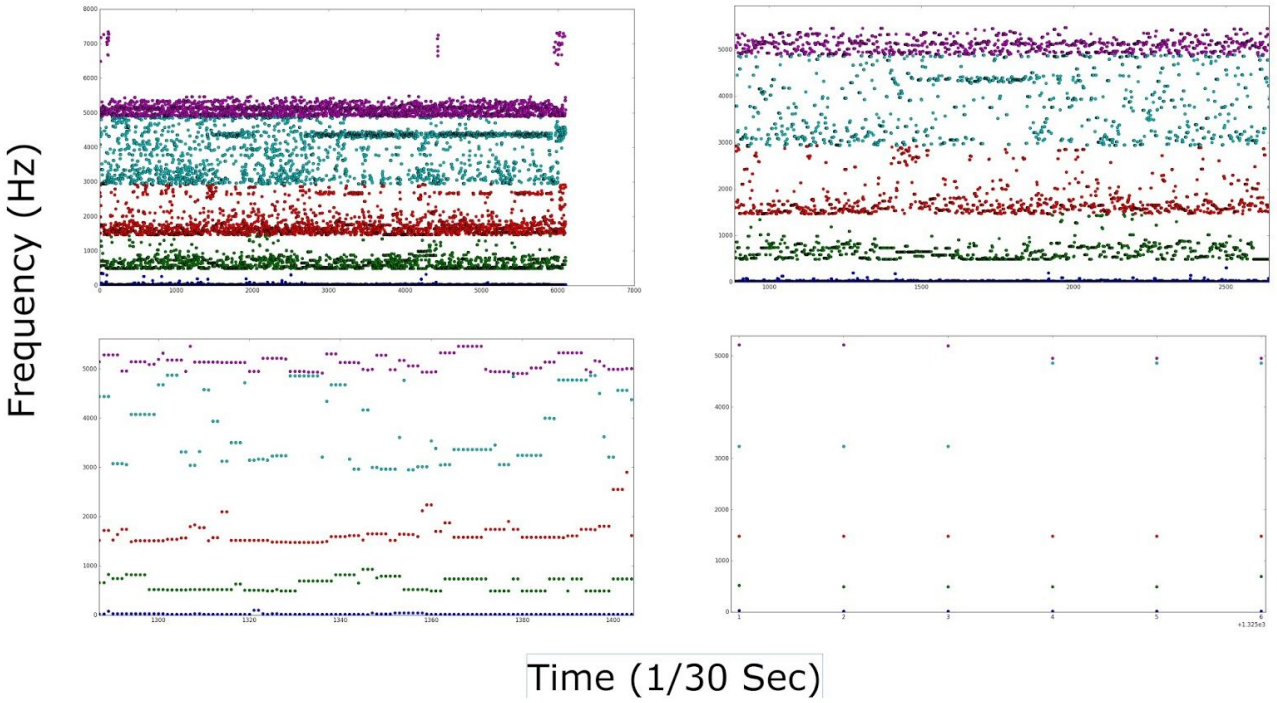
Shazam is an audio recognition application that allows for identification of recorded sound quickly and accurately. Before the creators of this algorithm released their method it was thought to be an impossible problem. The uses are mainly music identification but the algorithm can be used for other purposes such as radio monitoring and plagiarism detection in music. Shazam analyses the fourier transformation of recorded music. Through the use of constellation mapping, which uses the peaks of various frequencies in the music as identifying features in the music, the algorithm is able to quickly compare the constellations of the recorded sound against songs in their extensive database of music. The algorithm then ranks the matches and returns the best match. For the purpose of this report the shazam algorithm was used as inspiration to create a similar signal processing algorithm on a smaller scale.

Vertical Band Comparison:

When trying to identify music (or any audio for that matter), a common approach to doing so as to compare frequencies of the samples of audio. Generally, different sounds and noises are comprised of unique frequencies, which can be deciphered using a Fourier Transformation. If one were to then plot Frequencies versus time versus intensity, an average song may look like this:



The initial attempt of comparison of the samples and songs in the database used a vertical snapshot of the clip to compare to the full song. These vertical snapshots consisted of the max values of 5 frequency ranges of the song. The frequencies were found from taking the fourier transforms of the song at small windows. Best results were found using 1/30th of a second windows. This was a happy medium for frequency resolution while having a window small enough to see the fine changes in the music from one window to the next. These frequency ranges were split into ranges that may pick up different elements of the music such as bass, vocals, and specific percussion. When plotting the resulting data, we are given what is essentially the same spectrogram but at a much lower resolution that we can then utilize.



The process of matching was to take samples of the clip of the song and compare to the whole song using the numpy functionality of boolean comparison to quickly find matches and store them in an array holding the matches for a specific sample as a row of the matrix where all the frequencies matched. This comparison was done for all snapshots of the chosen sample density of the clip and stored as rows making the comparison array height clip sample amount and length number of full song windows. The amount of adjacent matches which represented data that was at the same spacing in the full song as well as the clip was the basis of comparison that would be used to ultimately compare matching.

Problems with Vertical Band Comparison:

The rigidity of the vertical band method made itself known in several ways. One if the recording was offset by less than a 30th of a second with the original song the maxes of the frequencies were affected enough to not match with the full match criteria. The full match criteria would really only provide a match from a clip that was sliced directly from the full song file. Which for the simple matching that we were working towards on this project it did work if the clip was created from the full song. But with any offset in the time or if the same clip was recorded on the phone and then read in and compared to the full song the matching process was unable to deal with the change in frequency values. This matching process was in need of improvement, some attempts were to create an acceptable tolerance of variation between the compared values this provided limited success. Ultimately the tweaking of the process to mitigate the shortcomings was not as worthwhile as a modification of the process at its root.

Results:

Hashing Full Song: hcym_full_2.wav
Done.

filename: hcymc2.wav
Total Matches 67.0
Adjacent count: 33
filename: hcymc2_60.wav
Total Matches 45.0
Adjacent count: 4
filename: hcymc2_record.wav
Total Matches 0.0
Adjacent count: 0
filename: hcym3.wav
Total Matches 21.0
Adjacent count: 2
filename: chittychitty.wav
Total Matches 0.0
Adjacent count: 0

This output shows the method at its final stages. Clips of the same song were used to test different shortcomings of the algorithm. hcym2.wav is a slice of the full song array. This one matches with almost perfect accuracy understandably due to exact data equality. hcym2_60.wav is a clip with the start time offset from the start time of the clip by a 60th of a second which is half of the distance that is used by the algorithm when hashing the full songs and the clip. Due to this shift some of the max values of frequencies are not equivalent which results in less matches. hcym2_record.wav is a clip re-recorded on a physical microphone. The algorithm was to rigid to be effective for this clip. hcym3.wav is a clip with noise added using the program audacity. The algorithm showed signs of matches but was very low. The algorithm was good with not generating matches for the incorrect song with the clip chittychitty.wav.

Code:

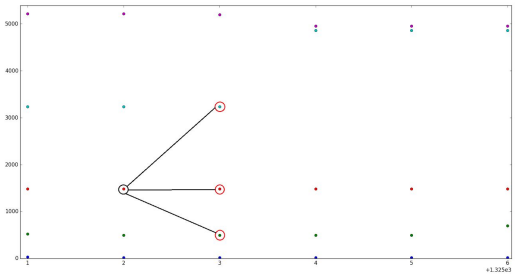
```
from numpy import max as nmax
from numpy import *
from scipy.fftpack import fft,ifft
from scipy.io import wavfile
import matplotlib.pyplot as plt
import warnings

warnings.simplefilter("ignore")

def shazam(filename,samplespersec = 30):
    rate,x = wavfile.read(filename)
    n = x.shape[0]
    if len(x.shape) == 2:
        x = (x.T[0])

    sections = []
    sec = float(samplespersec)
    dt = int(rate/sec)

    f = []
    width = 10*dt
    han = hanning(width)
    for j in range(0,len(x)-width,dt):

        i = x[j:j+width]

        frr = abs(fft(i*han))

        mid = len(frr)/2

        v = linspace(0,mid,16)
        v = v.astype(int)

        f.append([
            argmax(abs(frr[v[0]:v[1]])),
            argmax(abs(frr[v[1]:v[3]])),
            argmax(abs(frr[v[3]:v[6]])),
            argmax(abs(frr[v[6]:v[10]])),
            argmax(abs(frr[v[10]:v[15]]))
            ])

    return array(f)

def compiler(sound):
    data = []
    for i in range(1,len(sound)-10):
        data.append(hasher(sound, i))
    return array(data[:])

def hasher(sound, n, i=3, ac=3):
    hsh = []
    anchor = float(sound[n][3])
    hsh = [round(sound[n+1][0]/anchor,ac),
           round(sound[n+1][1]/anchor,ac),
           round(sound[n+1][4]/anchor,ac)]
    return hsh

def getmatchesvert(clip,full,sampletest = 50):
    comp = zeros((sampletest,len(full)))
    for i in range(sampletest):
        w = len(clip)/float(sampletest)
```

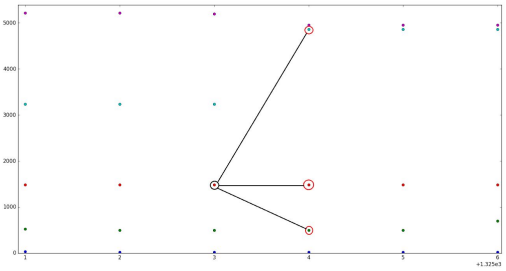
```
        comp[i,:] = all(clip[i*w,:] == full[:,:],1)
        set_printoptions(threshold='nan')

    return comp, len(clip)/sampletest

def evaluatevert(filenames, full_filename):
    print "Hashing Full Song: " + full_filename
    full = array(shazam(full_filename))
    print "Done."
    print ""
    full_data = compiler(full)

    for filename in filenames:
        print 'filename:',filename
        sound = shazam(filename)
        clip_data = compiler(sound)
        comp,spacing = getmatchesvert(clip_data,full_data)
        adjacentcount = 0
        print 'Total Matches' ,sum(comp)
        for i in range(shape(comp)[0]-1):
            if argmax(comp[i+1,:])-argmax(comp[i,:])==spacing:
                adjacentcount+=1
        print 'Adjacent count:', adjacentcount

clip_names = ['hcymc2.wav','hcymc2_60.wav','hcymc2_record.wav','hcym3.wav','chittychitty.wav']
full_name = 'hcym_full_2.wav'
for i, name in enumerate(clip_names):
    clip_names[i] = name
evaluatevert(clip_names, full_name)
```

Constellation Recognition:

The actual act of reading in the songs to be compared is very similar with a few small changes. The window that was used previously sliced up the song into uniform sections which started when the last one ended. An improvement on the windowing technique was to create windows of uniform size which incrementally move at a smaller width than the sample. This is done to address some of the previous issues of offset smaller than the window width that was experienced in the vertical band method.

The first step in the matching process is having a database of songs of which to pull a match out of. The hasher recall function is called at the start of the matching process it takes the filename and looks for an existing hash. If there is not one the function calls the shazam function which creates the frequency maxes for the specified ranges. The function then creates the hashes of this data. The process consists of making an integer that stores the values of the ratio of the constellation points divided by the anchor point rounded to 3 decimal points. This rounding allows for a small degree of variance in the constellations of the full song and the clip. What the hasher returns is a length 12 integer that stores the values of these constellation points. These hashes are then stored for every time step in the form of an array of integers.





The clip is then hashed and compared to the full song. The way this comparison works is by dividing up the full song into 10 second chunks which will all be compared to the clip. For each comparison of the clip and the equal length segment of the full song the matching hash values are counted. The counts are then added to a list of the total matches for each segment of the full song. The ratio of the max matches of a segment of the full song to the length of the clip is returned by the matcher algorithm. This matching is done for all of the full songs in the database. The song with the highest percentage of matches is the one the algorithm returns as the song.

Results:

Tested File: **Falling_For_You_Clip.wav**
Best Match : **Falling_For_You.wav**
Match % : 2.7%
Average % : 0.45%
0.186000108719 secs

Tested File: **Hang_Me_Up_To_Dry_Clip.wav**
Best Match : **Hang_Me_Up_To_Dry.wav**
Match % : 71.4%
Average % : 9.1125%
0.21599984169 secs

Tested File: **Here_Comes_Your_Man_Clip.wav**
Best Match : **Here_Comes_Your_Man.wav**
Match % : 0.7%
Average % : 0.25%
0.193000078201 secs

Tested File: **Hey_Clip.wav**
Best Match : **Hey.wav**
Match % : 2.7%
Average % : 0.5875%
0.176000118256 secs

Tested File: **I_Get_Around_Clip.wav**
Best Match : **I_Get_Around.wav**
Match % : 4.6%
Average % : 0.725%
0.199999809265 secs

Tested File: **Three_of_a_Perfect_Pair_Clip.wav**
Best Match : **Three_of_a_Perfect_Pair.wav**
Match % : 3.4%
Average % : 0.75%
0.185000181198 secs

With this constellation recognition method, we see rather decent results. The output reads as follows: "Tested File" indicates the file that was tested against the full database. For this output, we tested 10 second recorded clips using the laptop's microphone. Each filename with a "_Clip" appended to it means it is one of these recorded songs except "Hang_Me_Up_To_Dry_Clip", which is the only song we used Audacity to cut the clip directly from the original wave file.

The next line, "Best Match" indicates which of the 6 songs in our database that the algorithm felt the song was. Notice how all the clips in this example matched correctly, indicating that our method is rather accurate.

The next line "Match %" is the percent of individual constellations in the hash that matched over the most effective window (so that it is the max of all such percentages). Then, "Average %" is the average of all these matching percentages for the entire database. We notice how consistent these results are, and also see that "Hang_Me_Up_To_Dry" was matched 71.4%, which was exceptional compared to the typical 2% to 4% of the other songs. This is to be expected, given that this algorithm should work very well under ideal conditions.

Another thing to note is the time. This process took about .2 seconds at most per song tested. This is very fast, considering we are both hashing and testing the song to our database during this time. Assuming a linear increase, we could sort through almost 2,000 songs in a minute, which isn't bad on a personal computer.

A more detailed results shows what kind of ranges we are working with. We compared a clip of the song "Cannonball" to a database of 8 songs. It was correctly identified, being over 3 times as probable as the second choice.

```
Retrieving Cannonball_Clip.wav
Done.
Comparing to Cannonball_Clip.wav...

Retrieving Cannonball.wav
Results:        Cannonball.wav
Percent Match: 2.3%

Retrieving Falling_For_You.wav
Results:        Falling_For_You.wav
Percent Match: 0.0%

Retrieving Hang_Me_Up_To_Dry.wav
Results:        Hang_Me_Up_To_Dry.wav
Percent Match: 0.0%

Retrieving Here_Comes_Your_Man.wav
Results:        Here_Comes_Your_Man.wav
Percent Match: 0.7%

Retrieving Hey.wav
Results:        Hey.wav
Percent Match: 0.0%

Retrieving I_Get_Around.wav
Results:        I_Get_Around.wav
Percent Match: 0.0%

Retrieving I_Love_It.wav
Results:        I_Love_It.wav
Percent Match: 0.3%

Retrieving Three_of_a_Perfect_Pair.wav
Results:        Three_of_a_Perfect_Pair.wav
Percent Match: 0.0%

Tested File: Cannonball_Clip.wav
Best Match:  Cannonball.wav
```

Code:

```
from numpy import max as nmax
from numpy import *
from scipy.fftpack import fft,ifft
from scipy.io import wavfile
import matplotlib.pyplot as plt
import warnings
import json
import os
from dwnld import downloadAudio
from play import record, replay
import time

warnings.simplefilter("ignore")


def shazam(filename,samplespersec = 30):
    rate,x = wavfile.read(filename)
    n = x.shape[0]
    if len(x.shape) == 2:
        x = x.sum(axis=1)/2

    sections = []
    sec = float(samplespersec)
    dt = int(rate/sec)

    f = []
    width = 10*dt
    han = hanning(width)
    for j in range(0,len(x)-width,dt):

        i = x[j:j+width]

        frr = abs(fft(i*han))

        mid = len(frr)/2

        v = linspace(0,mid,16)
        v = v.astype(int)

        f.append([
            argmax((frr[v[0]:v[1]]))+v[0],
            argmax((frr[v[1]:v[3]]))+v[1],
            argmax((frr[v[3]:v[6]]))+v[3],
            argmax((frr[v[6]:v[10]]))+v[6],
            argmax((frr[v[10]:v[15]]))+v[10]
            ])

    return array(f)

def hasher(sound, n, ac=3):
    anchor = float(sound[n][2])
    hsh =   round(sound[n+1][1]/anchor,ac)*10**(ac*3+2)+\
          round(sound[n+1][3]/anchor,ac)*10**(ac*2+1)+\
          round(sound[n+1][2]/anchor,ac)*10**ac
    return hsh

def compiler(sound):
    data = []
    for i in range(1,len(sound)-30):
        h = hasher(sound, i)
        data.append(int(h))
```

```
    return array(data[:])

def matcher(clip_data, full_data):
    master = []
    m = len(clip_data)
    dm = m

    n = len(full_data)
    l = n-(n%m)
    dl = int(m/10)

    for j in range(0, n-m, dl):
        count = 0

        bl = 0
        full_test = full_data[j:j+m]
        for i in range(0,m,dm):
            bl = in1d(clip_data[i:i+dm],full_test[i:i+dm])
            count += len(bl[bl==True])
        master.append(count)

    return max(master)/float(m), m, n

def evaluate(filenames, full_filename):

    full_data = hashRecall(full_filename,'clips')
    print "Done."
    print "Comparing to " + full_filename + "..."
    print ""

    results = ["null",0.]
    for filename in filenames:
        clip_data = hashRecall(filename,'full')
        count, m, n = matcher(full_data, clip_data)
        count = int((count*1000))/10.

        print "Results:\t" + filename
        print "Percent Match:\t" + str(count)+'%'
        print ""
        if count > results[1]:
            results = [filename,count]
    print "Tested File: " + full_filename
    print "Best Match:  " + results[0]
    print "::::::::::::::::::::::::::::::::::::"
    print ""

def hashRecall(filename, drc):
    if os.path.isfile('hash\\' + filename+'_hash.txt'):
        print "Retrieving " + filename
        f = open('hash\\' + filename+'_hash.txt')
        data = json.load(f)
        f.close()
        return array(data)
    else:
        print "Hashing " + filename
        f = open('hash\\' + filename+'_hash.txt', 'w')
        full = shazam(drc+'\\'+filename)
        full_data = compiler(full).tolist()
        json.dump(full_data, f)
        f.close()
        return array(full_data)


def liveTest(url, name):
```

```
    print name
    downloadAudio(url, name)
    while True:
        raw_input('Press Enter when ready to record.')
        print "Recording in..."
        time.sleep(1)
        print "3"
        time.sleep(1)
        print "2"
        time.sleep(1)
        print "1"
        time.sleep(1)
        print "Recording..."
        record(name)
        if raw_input("Play?[Y/y]: ") in ['Y','y']:
            replay(name)
        if raw_input("Again?[Y/y]:") not in ['Y','y']:
            break

name = 'Cannonball'
url  = 'https://www.youtube.com/watch?v=fxvkI9MTQw4'

test_names = ['Falling_For_You',
              'Hang_Me_Up_To_Dry',
              'Here_Comes_Your_Man',
              'Hey',
              'I_Get_Around',
              'Three_of_a_Perfect_Pair']
clip_names = []
full_names = []

for content in os.listdir("clips"):
        clip_names.append(content)

for content in os.listdir("full"):
        full_names.append(content)

evaluate(full_names,name+'_Clip.wav')


for name in test_names:
    evaluate(full_names,name+'_Clip.wav')
    raw_input("Next?")
```

Conclusion:

Shazam was created to find a song match out of a database of millions of songs recorded on phone microphones with possibly large amounts of background noise. Our process shows the feasibility of this idea, albeit without being up to par with that of an industrial strength algorithm. While it is not on the same level as Shazam, our algorithm is capable of comparing the clip to a database of 1000 songs in about 30 seconds.

With this, the process has shown to be reliable in finding matches with clips that were taken directly from the data, offset by a smaller amount than the width of the samples of a given clip, recorded with phone microphones, and with background noise added electronically. This recognition power with an almost nonexistent rate of false positives show that this method has the capability of making accurate matches for a large database if the speed was increased.

The Fourier Transformation and, just as equally important, the Fast Fourier Transformation allows for this process to even be possible, so that such audio processing and recognition can be commercially viable. Audio recognition by signal processing is a vast field that could be explored in much greater depths than what is provided here. With more time and a larger scoped project, one could not only increase the speed and accuracy of this algorithm, but apply it in other fashions such as identifying voices, bird calls, type of objects, etc.

References:

"How Does Shazam Work - Coding Geek." *Coding Geek*. N.p., 23 May 2015. Web. 28 Apr. 2016. http://coding-geek.com/how-shazam-works/

“An Industrial-Strength Audio Search Algorithm” Avery Li-Chun Wang. http://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf