Cameron Wutzke
Billy Lin
Yesibao Muhamaiti

EE 341 Final Lab Report

Overview:

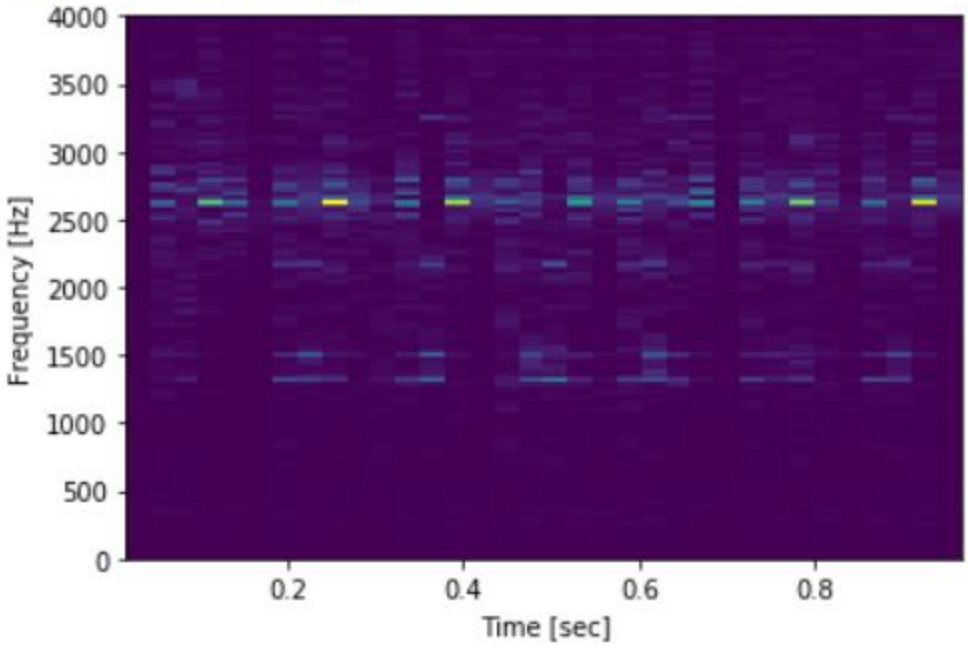
In this project, we created a small program that could identify songs with our manually-entered database. We collected some songs, found their fingerprints and saved them. When given an audio sample we will try to match fingerprints with our database to identify which song the audio sample is. We then will show the user the graph of the fingerprint of the audio sample and the fingerprint of the database we matched it with.

Methods:

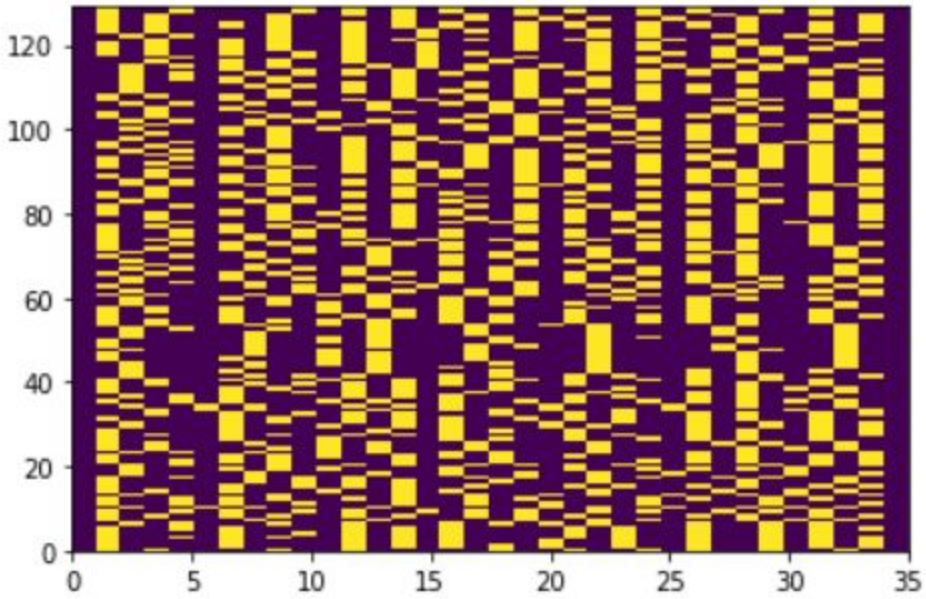
We use the following steps in our code to obtain a fingerprint of audio files:

1) Take the average of both channels and subtract the mean of this new data. Then, resample using a sampling rate of 8000
2) Create a spectrogram using signal.spectrogram with the new resampled data and sampling rate of 8000. This spectrogram gives us a 3D plot of magnitude/phase and frequency:

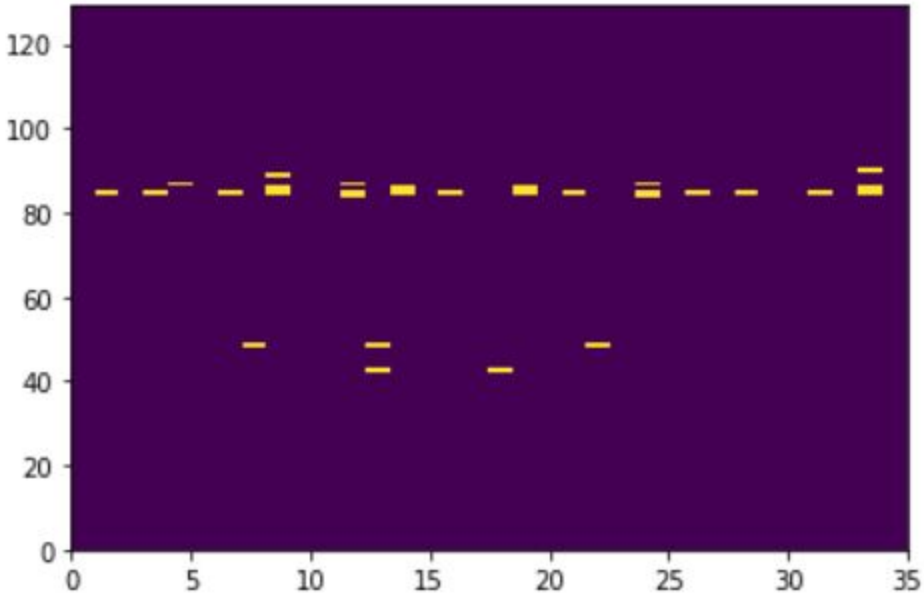
```
lengths of the 2 dimensions of Spectrogram:  (129, 35)
size of Spectrogram:  4515
```



3) From the returned spectrogram frequencies, we can find the local max values. These are values where the immediate right and left values are less than the current value. We set all local maximums = 1 and everything else = 0:



4) We then determine the threshold value by finding the 30th largest frequency value of the spectrogram which is *also* a local max. Once again we traverse through frequencies and set these frequencies which are higher than the threshold = 1 and everything else = 0:



We use the following helper functions to create/add fingerprints to our database:

getFingerprint: performs the steps outlined above

convert_to_pairs: takes in a song fingerprint and returns a table of proximate peaks

getWavList: returns a list of .wav files within a specific directory

load: reads and prints an entire file line by line

Add_to_table: adds the proximate pairs from “convert_to_pairs” along with a song ID number to a global hashtable

Using these functions listed the following steps are performed to match a song:

1) The global hashtable full of song IDs and song fingerprints must be loaded
2) The new audio file undergoes the process to obtain a fingerprint of this file
3) Frequency pairs are extracted from this new audio clip using the convert_to_pairs function mentioned above.
4) Time offsets are calculated and compared to the known offset within the database
5) The song with the most matches for a specific timing offset is identified to match the unknown new audio

Results:

We found it difficult to understand what was happening without visualization. Once we could see plots of the spectrogram, fingerprints, and local peaks, it made it easier to understand what was happening. The database could not be visually represented which made it difficult to understand errors and highlight bugs. For this, we printed many lines that showed the process of comparing timing offsets.

Our match table that contains the time offset for a test .wav file “do.wav” and it should be with SongID 2:

[[], [0.0, 0.0, 0.0, 0.0, -18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -8.0, 0.0, 0.0, -2.0, 0.0, 0.0, -6.0, 2.0, 0.0, 8.0, 0.0, 6.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [], [3.0], [0.0, -8.0, -16.0, 20.0, -10.0, -2.0, 10.0, 2.0, 5.0, 14.0, 3.0, 5.0, 5.0], [-3.0]]

As you can see, the second component of our match table have tons of time offset 0.0, and it’s going to be match. When we checked our directory of wav files, it was 02 value. So we succeeded.

Limitations:

We had problems loading .mp3 files to python. There is another package we must install to convert .mp3 files to .wav so the audio clips must be .wav format. This should not be an issue for users because plenty of online tools exist to convert .mp3 to .wav for free.

Based on how we average two channels from our audio files, audio files must have greater than 1 channel. If the audio file has greater than two channels, only the first two channels will be averaged which may give unexpected results.

Moreover, the operation number of our code is high, which means that if we build a very large database or receiving a very long wave file as an input, it might take a long time to generate a result.

Conclusion:

As for our team, we learned about audio processing to a deeper level and we put what we learned from the course into actual use, which is truly helpful to us. Even though there are hard tasks and the outbreak of coronavirus at the same time, our team still found a way to collaborate with each other and we think it helps us to build a team spirit and open-minded personalities.

As for the lab itself, this method of comparing audio fingerprints works well on a small scale. If we were to create a database as large as Shazam, with basically infinite songs, this method would not work well. The audio fingerprints do a good job compressing data from audio files, but comparing the unidentified fingerprint to every fingerprint in a database as large as Shazam's would take too much time. Perhaps algorithms could be created to limit the comparisons based on a song's tempo or popularity. So that our shazam can be better!