Audio Recognition

Database 4660
Wendy Osborn
University of Lethbridge, Fall 2016

Wyatt Trombley
Tyler Churchill

We decided to investigate the concepts behind the emerging field of audio recognition. We have prepared a program which utilizes basic audio fingerprinting technology. In general terms, the program converts audio files into a unique fingerprint. No two songs should have the same fingerprint, unless they are identical. We fingerprint the recorded audio and compare it to our database of fingerprint data. The main challenge in audio fingerprinting is the way you construct the fingerprint, and how efficiently you can store the fingerprint database.Initially we planned to implement two diverse fingerprinting strategies. However, we did not account for the massive overhead and undertaking that exists with audio data extraction. We instead opted for a much more realistic goal of a single “working” implementation.

Data Extraction

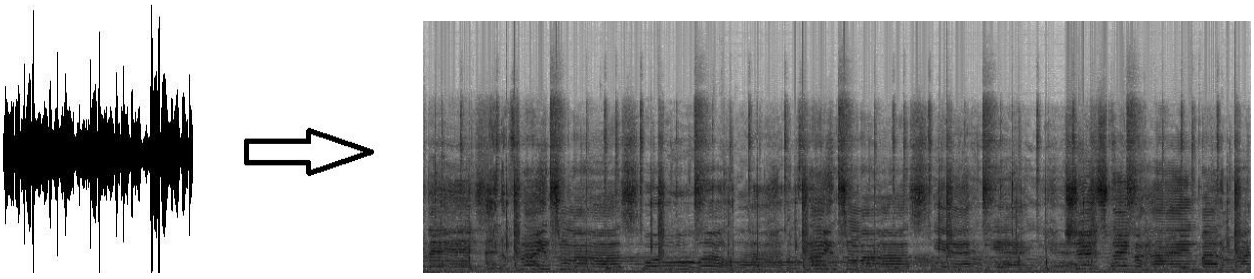


Figure 0 : The audio data from the wav file (time-domain) versus the data from the fourier transform (frequency-domain/spectrogram)

The audio recognition process begins by constructing a bank of audio features from each songs in our database. These songs are the ones we are trying to match a piece of recorded audio

with. The data from the .wav files are originally stored in time-domain. We are more interested in the defining features of audio and time domain data does not give us enough resolution in determining key features. In order to make the data usable we need to run a Fourier Transform on the audio data which gives us frequency-domain data (*see figure 0*). All audio in our database is transformed into 8 bit data, which gives us 256 units of frequency per frame. Each second of audio can contain many frames and a frame is the smallest unit of “time” we consider. (A four minute song can have approximately 20,000 frames)

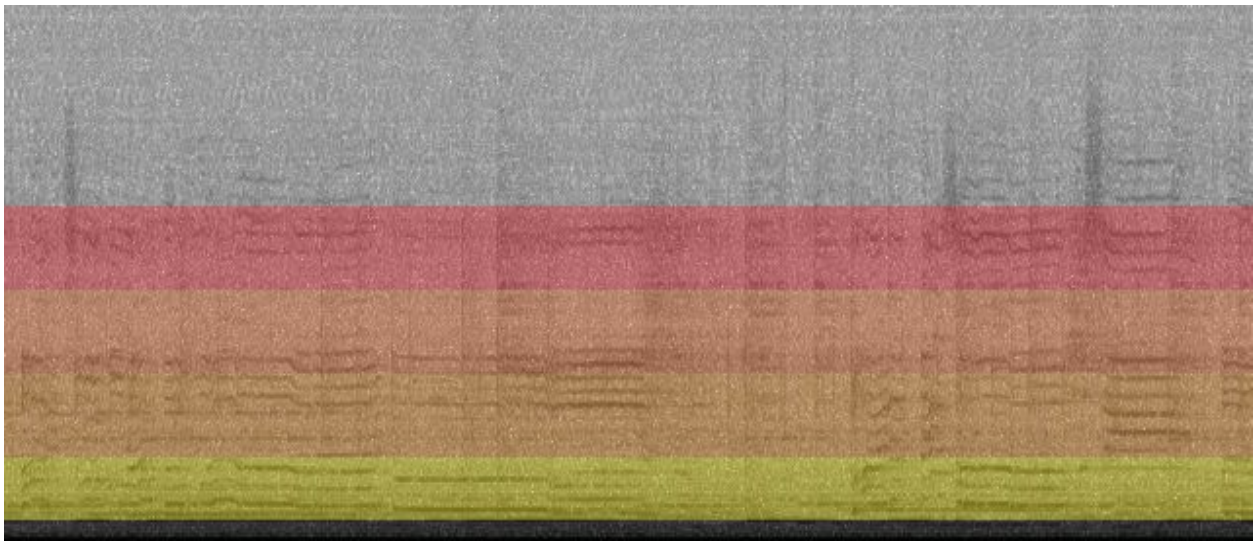


Figure 1 : Visual representation of the filter bank

The next step is to begin extracting key points or features from the spectrogram, which we will store in a database so we can later match key points from the recorded audio. From the Fourier transform data we want to select only the most important points from the songs. We decided to process every frame through a filter bank. We want to know which magnitude is the most dominant in a given range of frequency units. The frequency units range from 1-256, we pick four arbitrary ranges of these units and assign a high score to a specific frequency unit based on the magnitude of that frequency unit. We end up with 4 points for every frame of each song.

The data for these points is represented as an array of integers. For example one frame may give us the values 30, 45,85,105. This indicates that the dominant frequency units are 30,45,85,105at that specific frame. Now that we have these key magnitude values, we translate them into a hash value which we can use to group like frames from various songs. We use the keyframe digits as the hash key for each frame. Using the example points from the previous paragraph we would get 105854530 as our hash.(the magnitudes are simply reversed and appended.) Along with a hash key we also store a list of song points. A song points contains the frame number and songID. Each hash corresponds to a list of song points. Every song point in the list hashed to the same hash value as one another. These values are all stored in our database where hash is the primary key.

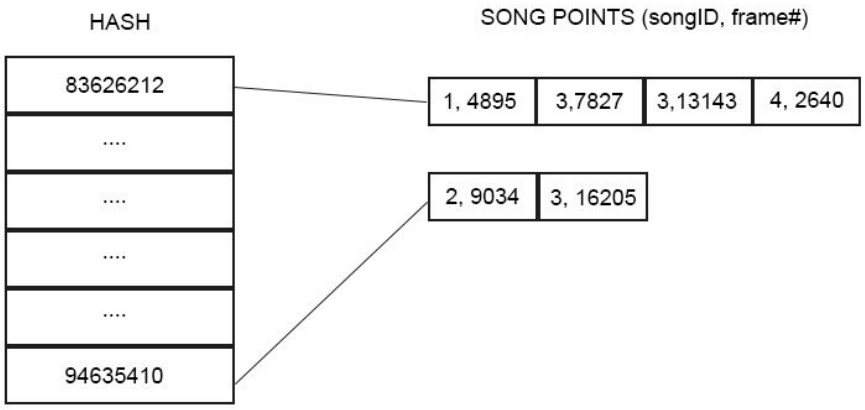


Figure 2 : (Table of hashes in our database) 836263212 has 4 frames of data that hashed to it. Song 1 at frame 4895, song 3 at frame 7827 and 13143 and song 4 at frame 2640

Searching by Key Points

We begin our search by recording some audio from the microphone. Once the audio has been recorded we start extracting its keypoints, just as we did with the songs in the database. Once completed we compare the hash values of the recorded points against the hash values in our database. If we find a matching hash it means we have some similar looking audio at that frame. Using sheerly the number of hits as a heuristic in our search yields very poor results. We instead need to look at the difference in frame numbers that the matching song point took place at. After searching through all matched song points, we can see that the song with the largest number of matched frames with corresponding offsets is our best guess to what song was recorded.

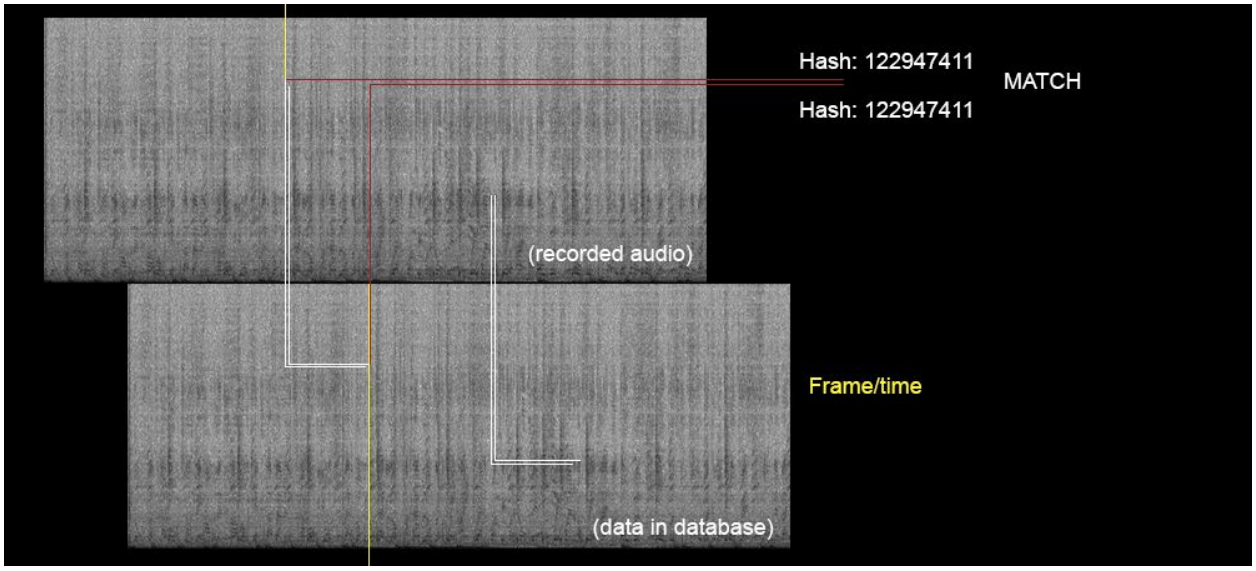


Figure 3: Visual representation of the search process with time offset

Results and Findings

Taking the keypoints from all of the songs takes roughly 0.9 - 1.2 seconds per song on our test machine.To counteract this processing time we opted not to rebuild the database every time the program is ran, and instead saved data to a flat file. The database can be loaded/saved from file, bypassing a majority of the processing time. Loading and parsing the data into our database takes roughly 1.5 seconds for the entire dataset.

Our goal when beginning the project was to have search return the correct song when a recording is played with minimal background noise and interference. Due to the large amount of variables when recording off a mic, we used an audio file already in the dataset as initial test data. This ensures that things like background noise and sample rates won't affect results.

```
FILENAME: TheLottery-CircaSurvive.wav
samplesize: 1024
 April-Tesseract.wav
Score: 51
 BattleRoyal-DopeDOD.wav
Score: 14
 CantSleep-VanikK-flay.wav
Score: 33
 FlyingtoMars-Foriegn Beggars.wav
Score: 50
 GetOut-CircaSurvive.wav
Score: 24
 KeyholeintheSky-Rishloo.wav
Score: 57
 Modus-ForeignBeggars.wav
Score: 19
 PerfectMoment-Tesseract.wav
Score: 24
 PolarBear-Puscifer.wav
Score: 35
 StrangeTerrain-CircaSurvive.wav
Score: 30
 TheLottery-CircaSurvive.wav
Score: 24427
 TotheWolves-MandroidEchostar.wav
Score: 28
 ViolentBounce-ISeeStars.wav
Score: 36


Best guess TheLottery-CircaSurvive.wav
(search took1707ms)
```

Figure 4:

Figure 4 shows that we can can consistently match these audio files to our database with 100% accuracy. We match every frame of the correct song (about 20,000) and only have about 10-50 false matches. The time taken is a lot longer than a search using recorded data, because we are comparing an entire four minute song to the database as opposed to a few seconds. This demonstrates that the search algorithm is working properly. Shazam claims to have a similar rate of false positives (around 0.1% - 0.01%)**[2]**.

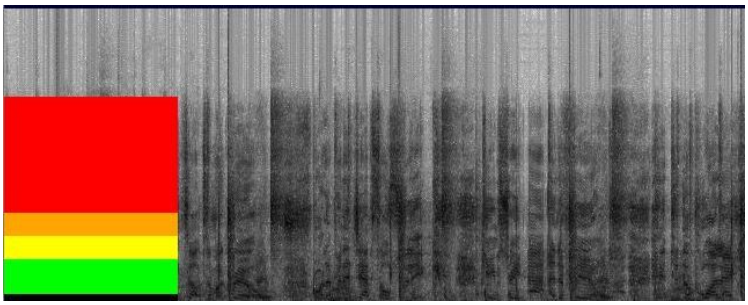
Now that we knew our algorithm will allow us to accurately search for a perfectly recorded audio data, we started to pass in live recorded audio from an external microphone. Figure 5 searched with a 3 second recording and figure 6 searched with a 12 second recording. Both figure 5 and 6 started recording at the same time in the song. The search time that is displayed combines the time it took to extract keypoints, and match the keypoints in the database. Our time taken still scales logarithmically with the size of our recording. The size of our database does not impact the search time due to the use of a hash function as the primary storage mechanism. Regardless if we look through 10 songs or 10,000 songs we still get constant time when searching our database. **[2]**

```
Started recording...
Stopped recording...
samplesize: 1024
 April-Tesseract.wav
Score: 3
 BattleRoyal-DopeDOD.wav
Score: 2
 CantSleep-VanikK-flay.wav
Score: 3
 FlyingtoMars-Foriegn Beggars.wav
Score: 2
 GetOut-CircaSurvive.wav
Score: 2
 KeyholeintheSky-Rishloo.wav
Score: 2
 Modus-ForeignBeggars.wav
Score: 2
 PerfectMoment-Tesseract.wav
Score: 2
 PolarBear-Puscifer.wav
Score: 12
 StrangeTerrain-CircaSurvive.wav
Score: 3
 TheLottery-CircaSurvive.wav
Score: 2
 TotheWolves-MandroidEchostar.wav
Score: 3
 ViolentBounce-ISeeStars.wav
Score: 2


Best guess PolarBear-Puscifer.wav
(search took79ms)
```

```
Started recording...
Stopped recording...
samplesize: 1024
 April-Tesseract.wav
Score: 6
 BattleRoyal-DopeDOD.wav
Score: 3
 CantSleep-VanikK-flay.wav
Score: 4
 FlyingtoMars-Foriegn Beggars.wav
Score: 5
 GetOut-CircaSurvive.wav
Score: 4
 KeyholeintheSky-Rishloo.wav
Score: 6
 Modus-ForeignBeggars.wav
Score: 5
 PerfectMoment-Tesseract.wav
Score: 3
 PolarBear-Puscifer.wav
Score: 34
 StrangeTerrain-CircaSurvive.wav
Score: 5
 TheLottery-CircaSurvive.wav
Score: 3
 TotheWolves-MandroidEchostar.wav
Score: 4
 ViolentBounce-ISeeStars.wav
Score: 4


Best guess PolarBear-Puscifer.wav
(search took150ms)
```

Figure 5 **Figure 6**

After seeing these results, we want wanted to see how the program would handle a moderate amount of background noise. We then searched by a recording which had a background conversation in it (Figure 7). To our amazement, the algorithm was still able to match the correct song.



```
Started recording...
Stopped recording...
samplesize: 1024
 April-Tesseract.wav
Score: 2
 BattleRoyal-DopeDOD.wav
Score: 2
 CantSleep-VanikK-flay.wav
Score: 2
 FlyingtoMars-Foriegn Beggars.wav
Score: 16
 GetOut-CircaSurvive.wav
Score: 2
 KeyholeintheSky-Rishloo.wav
Score: 2
 Modus-ForeignBeggars.wav
Score: 3
 PerfectMoment-Tesseract.wav
Score: 2
 PolarBear-Puscifer.wav
Score: 2
 StrangeTerrain-CircaSurvive.wav
Score: 3
 TheLottery-CircaSurvive.wav
Score: 2
 TotheWolves-MandroidEchostar.wav
Score: 2
 ViolentBounce-ISeeStars.wav
Score: 2

Best guess FlyingtoMars-Foriegn Beggars.wav
(search took113ms)
```

Figure 7 (positive match with background noise)

Conclusion

This project was extremely eye opening and overall a success. A lack of digital audio skills on our end resulted in our search algorithm being somewhat limited. Algorithms listed in **[1]** are able to handle files at different levels of compression. Currently our implementation is limited to extracting fingerprints from .wav files. Wav files are uncompressed which makes it easy to extract audio data. The downside is that most music is saved and used in the mp3

format so we have to first convert all of the files we wish to match against to the wav format. Due to the restriction on filetype we were unable to compile a large database of music to search against. We had to manually convert any test data resulting in a library of only a dozen or so songs. This resulted in our audio files adding up to 532Mb of data, which when extracted and saved to our fingerprint format resulted in a database 5Mb in size. Having a more robust program that is able to handle different file types would massively increase usability. Generating histograms and other visual data would have simplified troubleshooting errors in the algorithm.

Having the search begin as soon as the recording begins was something that would have been ideal to include. However, managing multiple threads proved to be very time consuming. The algorithm in **[2]** is capable of adapting to distortion and corruption of audio files as well as a large degree of background noise, This is another improvement that could be made to our implementation. Tweaking the frame resolution as well as the frequency filter bank altered the results dramatically. We tested converting the time unit from frames to seconds, the loss in resolution caused a dramatic increase in the amount of false matches we received. The filter bank values make or break the algorithms ability to properly match audio. The extreme low and high ends of the spectrometer are relatively worthless as keypoints due to the large amount of noise. In our testing, the best results were received when a focus was put on the middle range frequencies.

REFERENCES

[1] Doets, P. J. O., M. Menor Gisbert, and R. L. Lagendijk. "On the Comparison of Audio Fingerprints for Extracting Quality Parameters of Compressed Audio." *Security, Steganography, and Watermarking of Multimedia Contents VIII*, 2006. doi:10.1117/12.642968. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.2609&rep=rep1&type=pdf)

[2] Wang, Avery Li-Chun. "An Industrial-Strength Audio Search Algorithm." (https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf)