Robust Audio Fingerprinting Using Combinatorial Hashing and Temporal Offsets

Luke M. Craig

December 9, 2018

1 Abstract

We have implemented the algorithm described in *An Industrial-Strength Audio Search Algorithm* [1] and reproduced the results. We have also elaborated on sub-component algorithms that the original paper did not specify.

2 Table of Contents

3 Table of Figures

4 Introduction

4.1 Motivation

The “Shazam” App for mobile phones can listen to a short recording of music being played in a noisy environment and quickly return the song’s artist and title to the user. The developer of Shazam, Avery Wang, published his algorithm in 2003 [1]. This algorithm is also what websites like YouTube probably use to automatically detect copyright infringement in videos with background music. This algorithm is only useful for detecting exact matches. It does not find similar sounding songs or different versions of the same song.

The naïve solution to this problem would be to store a spectrogram of every possible song audio file in a database and then slide the spectrogram of the audio to query across every time segment of every song in the database keeping track of the minimum cosine distance, and then returning the song with the overall minimum cosine distance. This, of course, is not feasible and would possibly take years to match a single clip.

5 Preliminaries

5.1 Audio data

Raw uncompressed audio data is digitally represented as a series of signed integers that represent an electrical wave's amplitude between 100% and -100% as it oscillates over zero. With 16-bit audio, which is what audio CDs use, 100% is represented by 32,767. These numbers are sampled from the electrical audio signal at a fixed rate. Audio CDs have 44,100 samples per second and, due to the Nyquist Sampling Theorem, can store sound frequencies of half that, or 22.05kHz. When audio playing software reads this audio data, the signed integers are interpreted as voltage for an electrical signal which is sent to the computer's speakers and determines how far their speaker cone is pushed in and out. The pushing and pulling of speaker cones creates compressions and rarefactions in the air, which our brains perceive as sound.

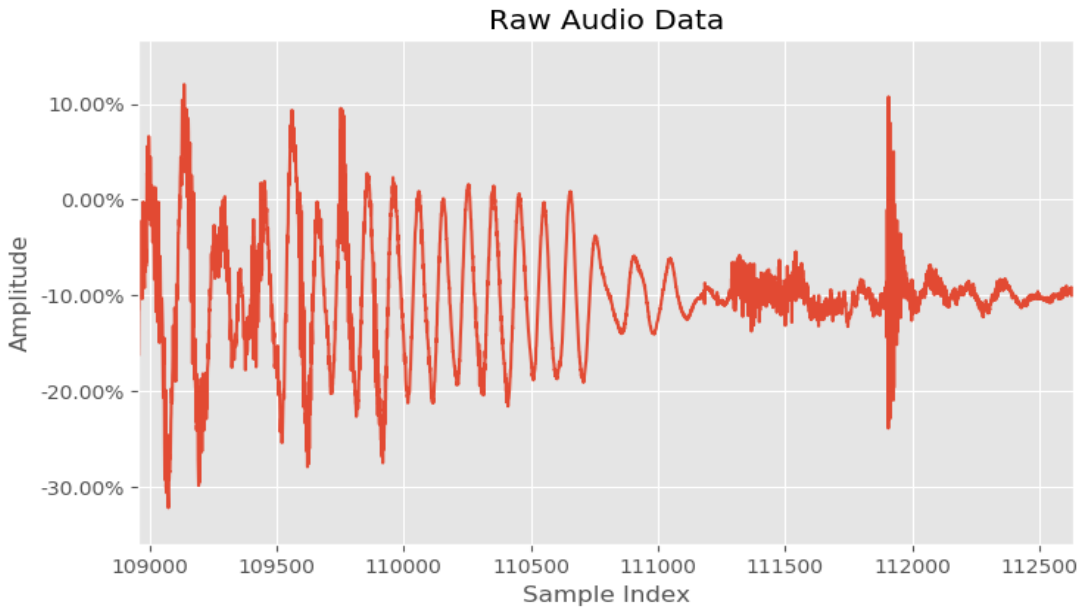

Figure 5-1: Raw Audio Data

5.2 Fast-Fourier Transform

Audio data can be transformed from the raw time domain into the frequency domain using the FFT. In this representation, the temporal information is traded for the magnitude and phase of an equally spaced number of frequency bins. The number of samples in the input audio data determines the number, and thus width, of the frequency bins. The result of transforming audio data into the frequency domain is called a spectrum.

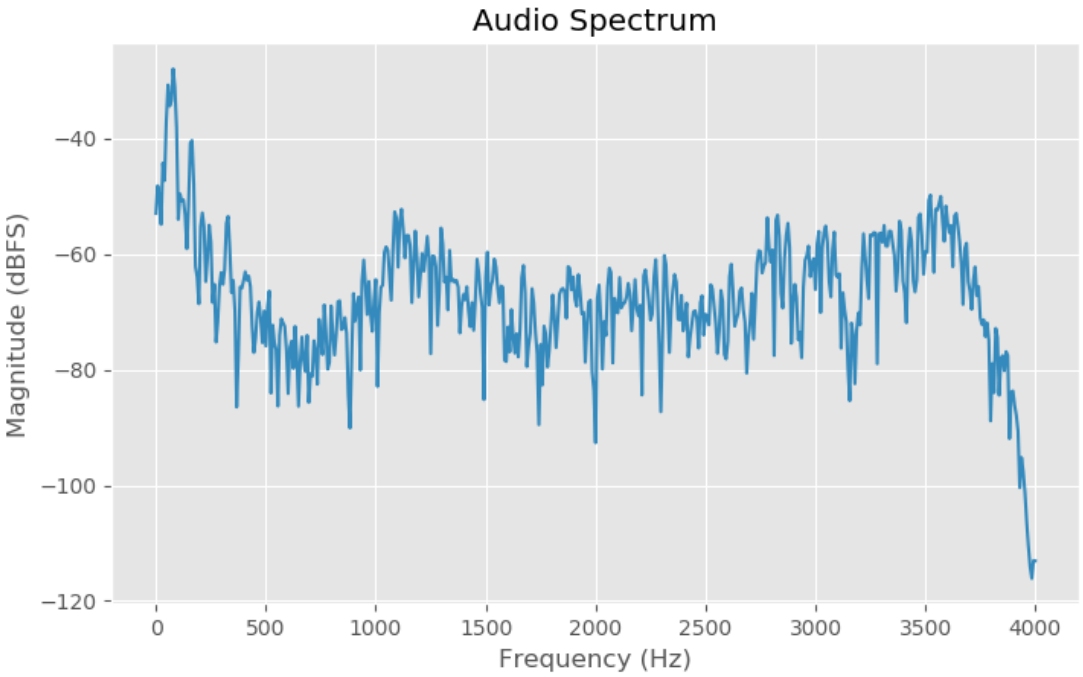

Figure 5-2: Audio Spectrum

5.3 Spectrogram

To prevent the loss of the temporal information, the Short-Time Fourier Transform, or STFT, can be used. With the STFT, the original data is divided into a series of overlapping segments. Each segment is then transformed into the frequency domain. By placing the segments in a 2D matrix we have a time-

frequency representation of the audio. The visual representation of the magnitudes is called a spectrogram.

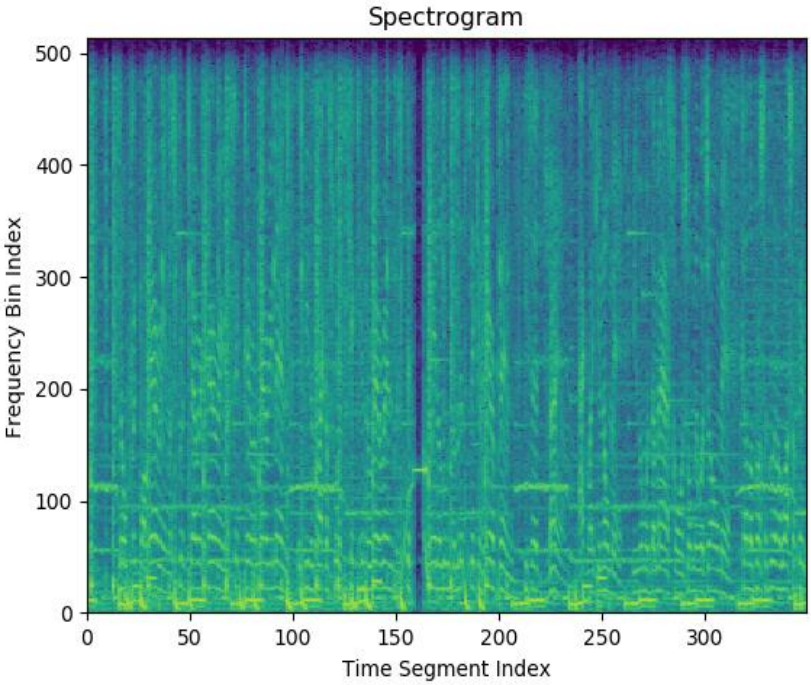

Figure 5-4: Audio Spectrogram

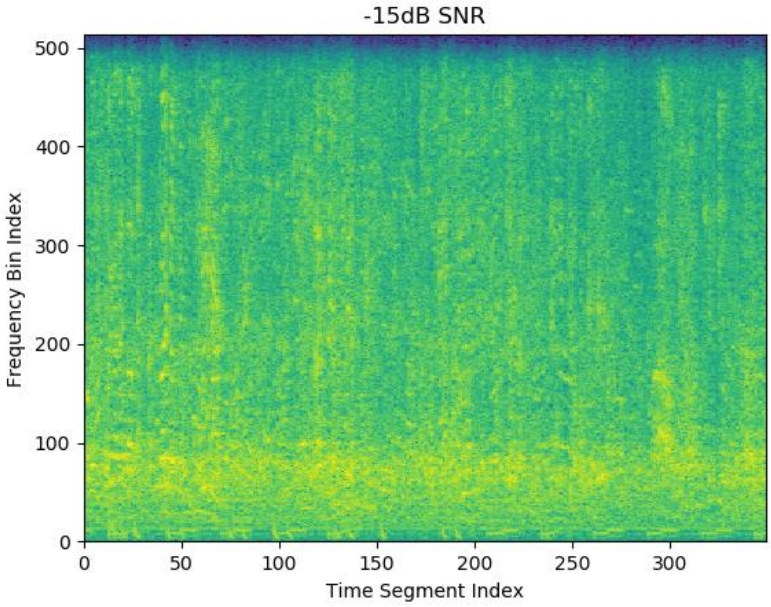

Figure 5-3: Noise Added at a -15dB SNR

5.4 Signal-to-noise ratio

Signal-to-noise ratio is the ratio of the power (measured as root-mean-square in this paper) of the clean audio signal with the power of the noise audio signal. It is represented in decibels (dB) which were calculated as $20 * \log_{10}(\frac{s}{n})$, where s is the RMS power of the clean signal and n is the RMS power of the noise signal. Figure 5-3 shows the same audio as Figure 5-4 but with noise added at a -15dB SNR. The difficult for a human to visually match those two figures illustrates the difficulty for the algorithm to correctly identify the original recording at this SNR.

6 Algorithm

6.1 Combinatorial Hash Generation

Before a song can be identified by a user, the original full recording must be analyzed and inserted into a database. First, the audio is converted into a spectrogram. Then the spectrogram is converted into a sparse representation by performing two-dimensional peak detection on it.

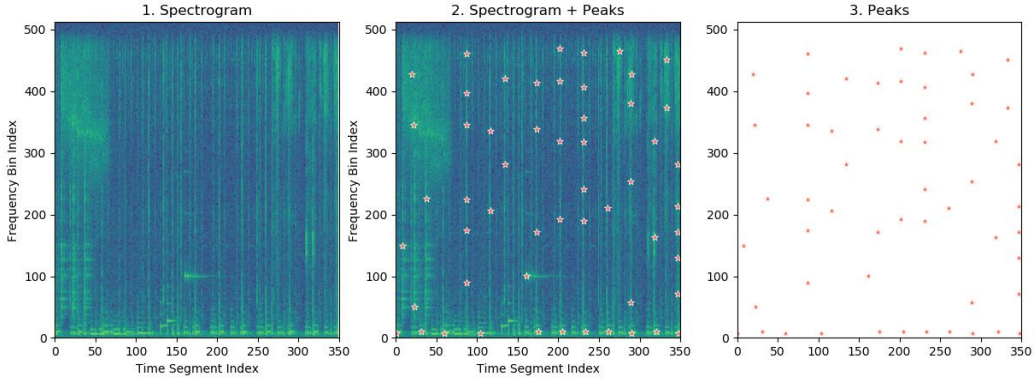

Figure 6-1: Spectrogram to Sparse Peak Representation

After generating the peaks and discarding the spectrogram, each peak is paired with neighboring peaks that fall within a "target zone," relative to a peak's position. Each pairing is how a combinatorial hash key is formed. The hash function is simply concatenating three 10-bit numbers into a 30-bit number. The first 20-bits are the frequency bin of the first peak and the frequency bin of the second peak. The last 10-bits is the time-delta between the two peaks. The value that this hash key looks up is a 16-bit number

representing the song ID and another 16-bit number representing the time offset between the first peak in the pair and the beginning of the recording.

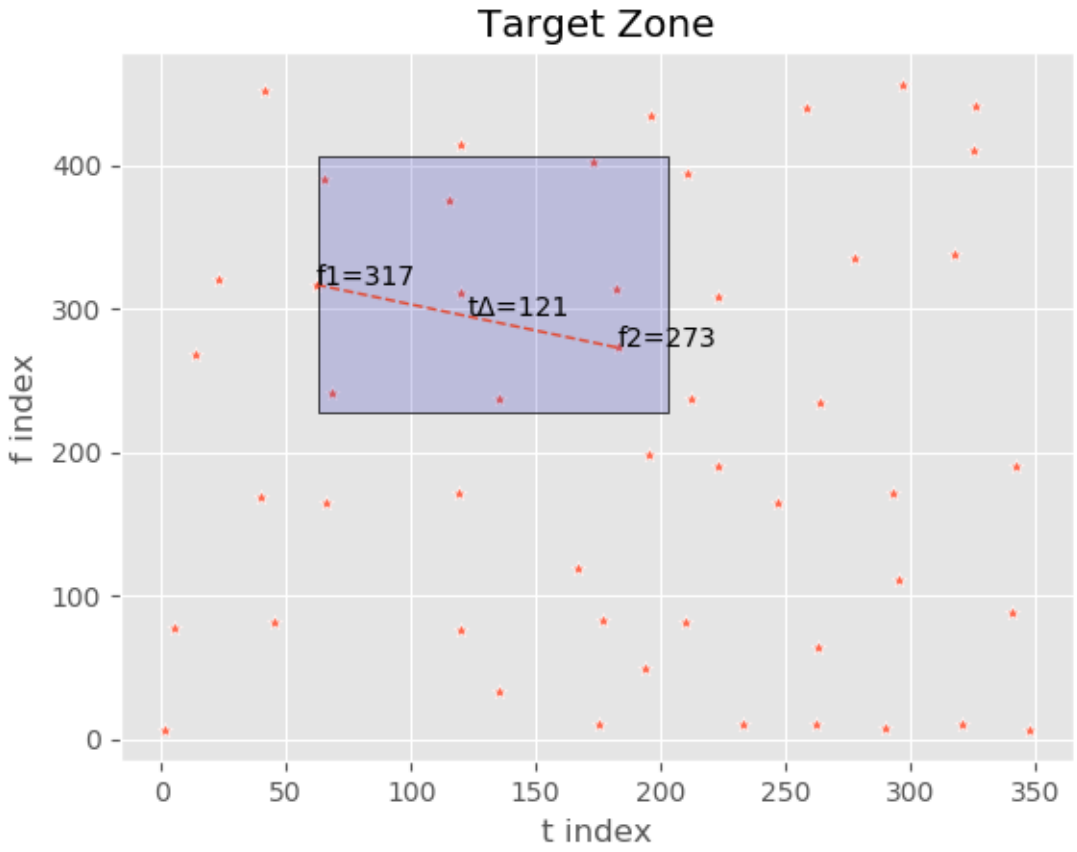

Figure 6-2: Target Zone and Hash Generation

Each song can have thousands of fingerprints, depending on the desired density of the spectrogram peak detection and the target zone size. In addition, each fingerprint hash key may have many collisions with fingerprints for different songs or fingerprints from the same song at different temporal locations. For example, an individual pop song's choruses are likely to be similar and generate similar fingerprint hashes.

6.2 Song Matching

Once a song has all its fingerprints in the database, it can be identified from a small clip, from any point in the song, even when background noise is present. The process for generating the fingerprint hash keys is the same as in the original analysis process. Since the algorithm needs to work for a clip no matter where in the song its located, these hash keys are paired with the time offset of the fingerprints first peak and the beginning of the clip, rather than the beginning of the song. In addition, since we don't know the song, the song ID is not present in the local fingerprints either. Once the clip's fingerprints have been generated, the database is queried for all the fingerprints with the same hash key. If a song is correctly matched, the value of the local and remote temporal offsets of each fingerprint will be correlated with a slope of 1, as in Figure 6-3.

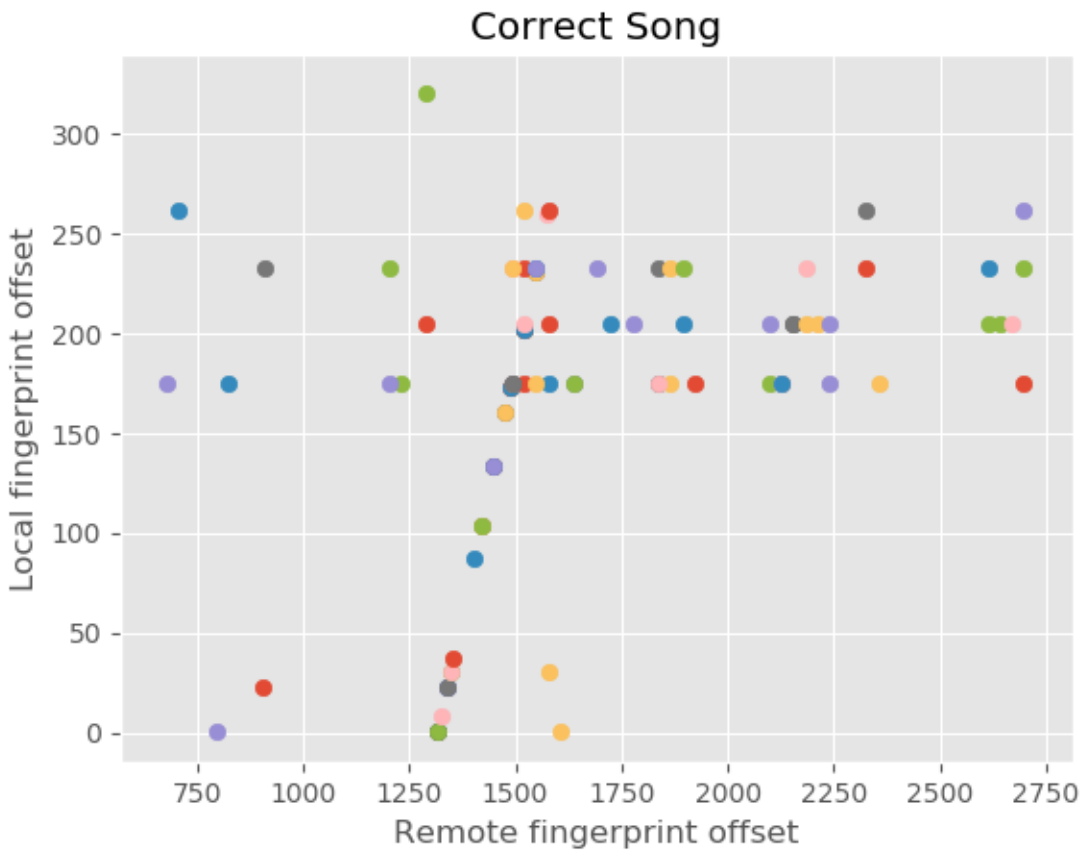

Figure 6-3: Scatterplot of Local and Remote Offsets - Linear Correlation

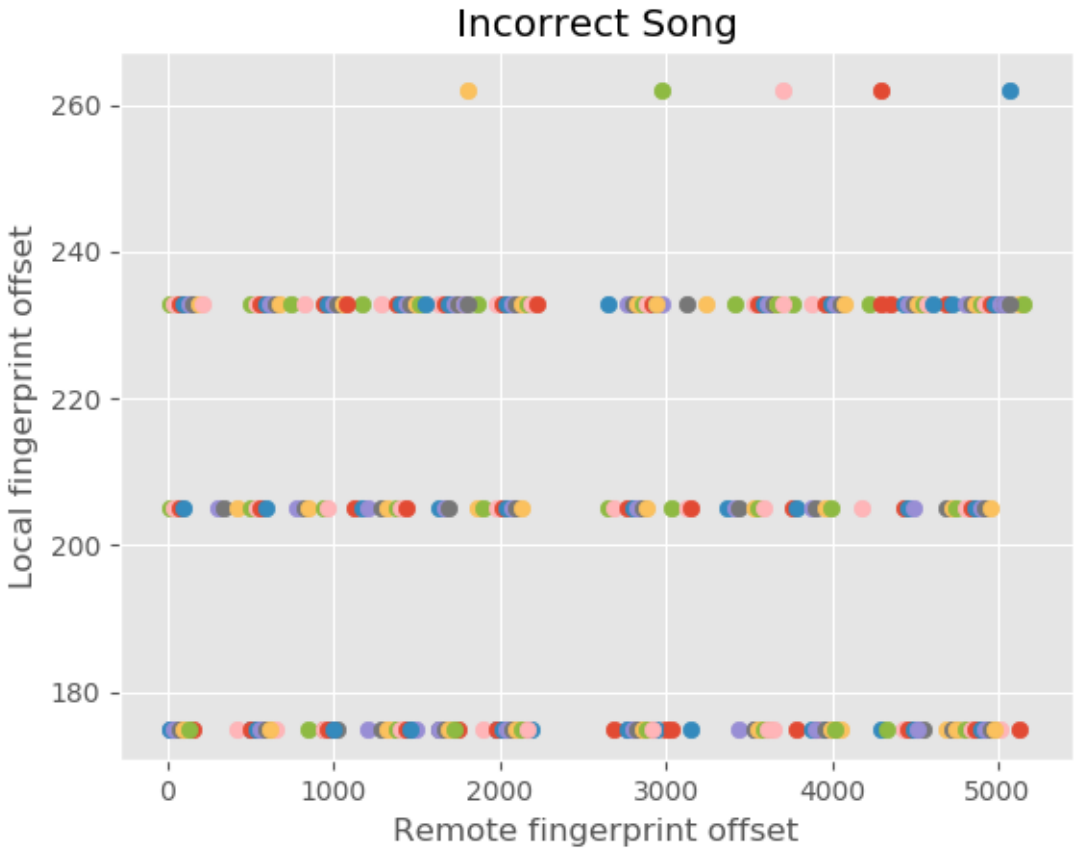

Figure 6-4: Scatterplot of Local and Remote Offsets - No Linear Correlation

The song in Figure 6-4 has the most matching fingerprints but there is no correlation in temporal offsets so we know it is not a true match.

In order to detect this linear correlation, a list of time deltas between each local fingerprint's and corresponding remote fingerprint's offsets is generated. If a song is a true match, then it will have several of these offset time-deltas that are equal. To determine the best match, each song ID from the set of song IDs in the matching fingerprints, has a histogram generated of these offset time-deltas. Peak detection is performed on each histogram and the histogram with the cluster of the highest peak is determined to be the matching song. Figure 6-5 shows the matching song has a clear peak at a time-offset delta of 1600 time-segments.

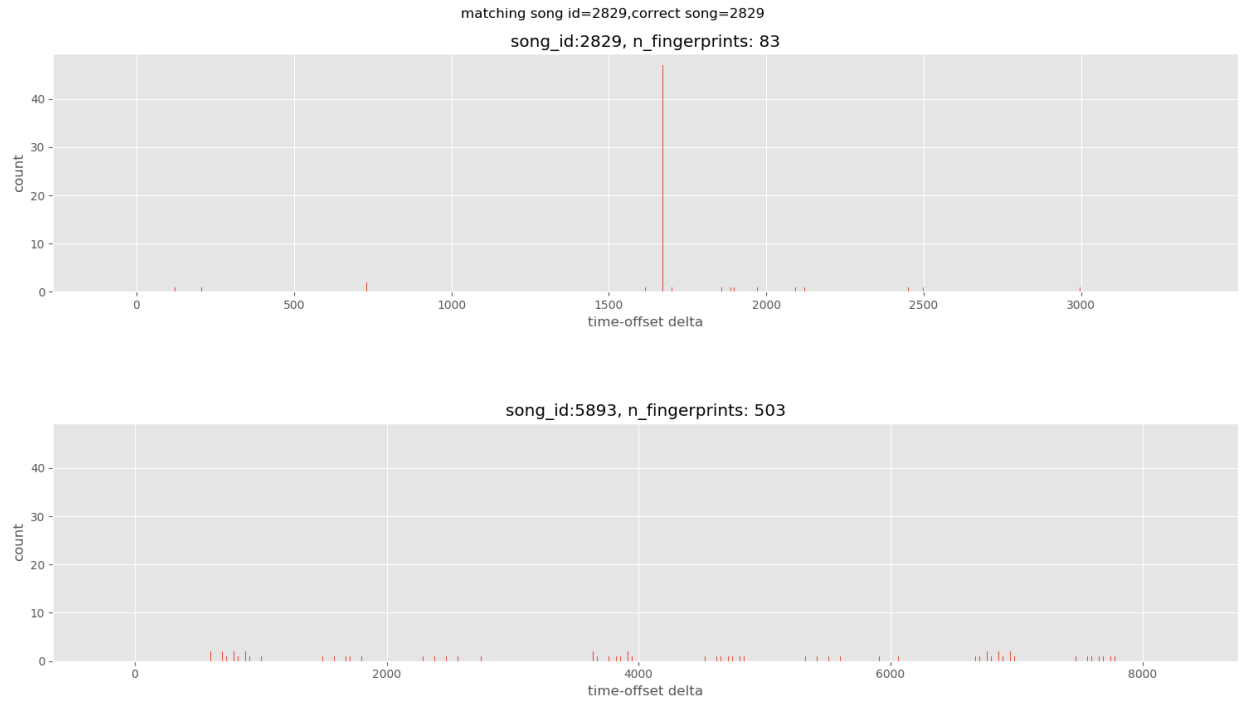

Figure 6-5: Histogram Peak of Offset Deltas

6.3 Robustness

The algorithm is robust against background noise because of the process of representing spectrograms by peaks. When there is background noise, the peaks are what will survive. In addition, because the peaks no longer store the magnitude information that the original spectrogram stored, they will be matched even if the additive noise causes the relative magnitudes in a peak-pair to be different than they were in the original clean recording. In addition, by representing the spectrograms by their peak locations, the storage data needed to represent a song is considerably reduced.

7 Implementation

7.1 Python

We implemented the algorithm in Python 3. One disadvantage to this is that Python's interpreter, CPython, uses a Global Interpreter Lock (GIL), to prevent threads from running in parallel. However, by

using the NumPy and SciPy and Pandas libraries, we can take advantage of compiled C code that does run sometimes in parallel, as well as vectorized SIMD processing in some cases.

7.2 Parameters

Wang did not specify the density criterion or the target zone dimensions they used to get their reported results. We decided to use what these appeared to be in their figures. We used a target zone of 6 seconds by 1400 Hz. The figure in the original paper suggests that the target zone should start over a second after the anchor peak but we decided to have it start immediately after the anchor peak so that recognition of smaller audio clips would be feasible. With this target zone size, on average the number of pairs-per-peak was 5, with a max of 9.

Our implementation does not let one specify a spectrogram peak density criterion directly, but the parameters we used (described in 7.3 below) led to an average of 4.46 peaks-per-second in the test set, with a minimum of 2.23 and a maximum of 6.5. Those values should be roughly the same from song to song, regardless of the length of the song.

7.3 Spectrogram Peak Detection

The original paper by Wang does not specify how they accomplished detecting the peaks in the spectrogram. They only state that, “candidate peaks are chosen according to a density criterion in order to assure that the time-frequency strip for the audio file has reasonably uniform coverage.” Our implementation detected peaks by using SciPy’s Maximum Filter function from their Multidimensional Image Processing package. This function takes our original spectrogram matrix and creates a new equally sized matrix where each pixel (or time-frequency bin in our case) is set to be the maximum value of a window, of which it is the center of. It implements Richard Harter’s description of the MINLIST

algorithm, which reportedly has a $O(n)$ running time for the 1D case, where n is the number of pixels, or time segments in our case, regardless of the filter size.

Figure 7-1 gives a one-dimensional visualization of this algorithm. The green window represents the sliding window, which in this case is set to a size of 6 time segments. As the window slides across the blue input, the red line is generated at the black midpoint of the window. It is set to the maximum value inside the green window. For the 2D spectrograms, this same process occurs but the sliding window is a

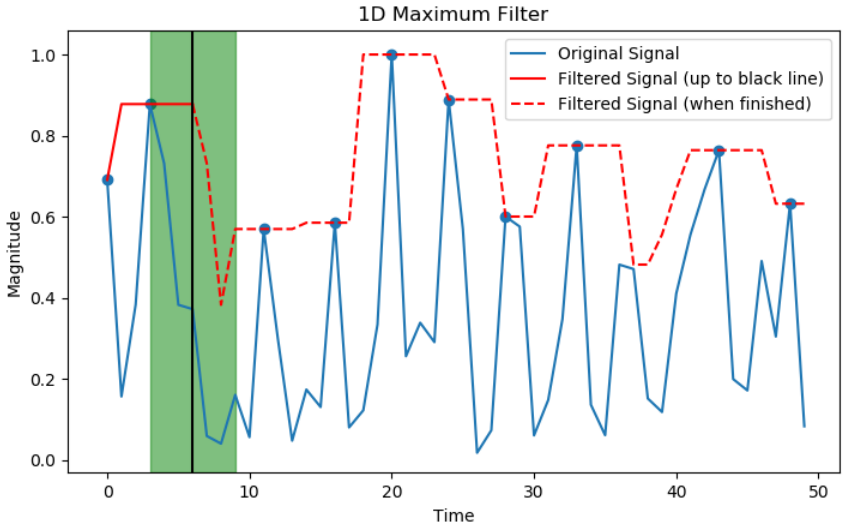

Figure 7-1: Visualization of a 1D Max Filter

rectangle of a fixed width for time and a fixed height for frequency. Because SciPy executes compiled C code, it is able to use true parallelized multithreading outside of the GIL. Therefore it was more efficient to use SciPy's function than coding this in Python directly. We finish detecting peaks by doing a Boolean matrix equality check between the filtered spectrogram and the original. Peaks are determined to be where the two are equal. In Figure 7-1 this is represented by the blue circles. In Figure 7-2 it is represented by orange stars.

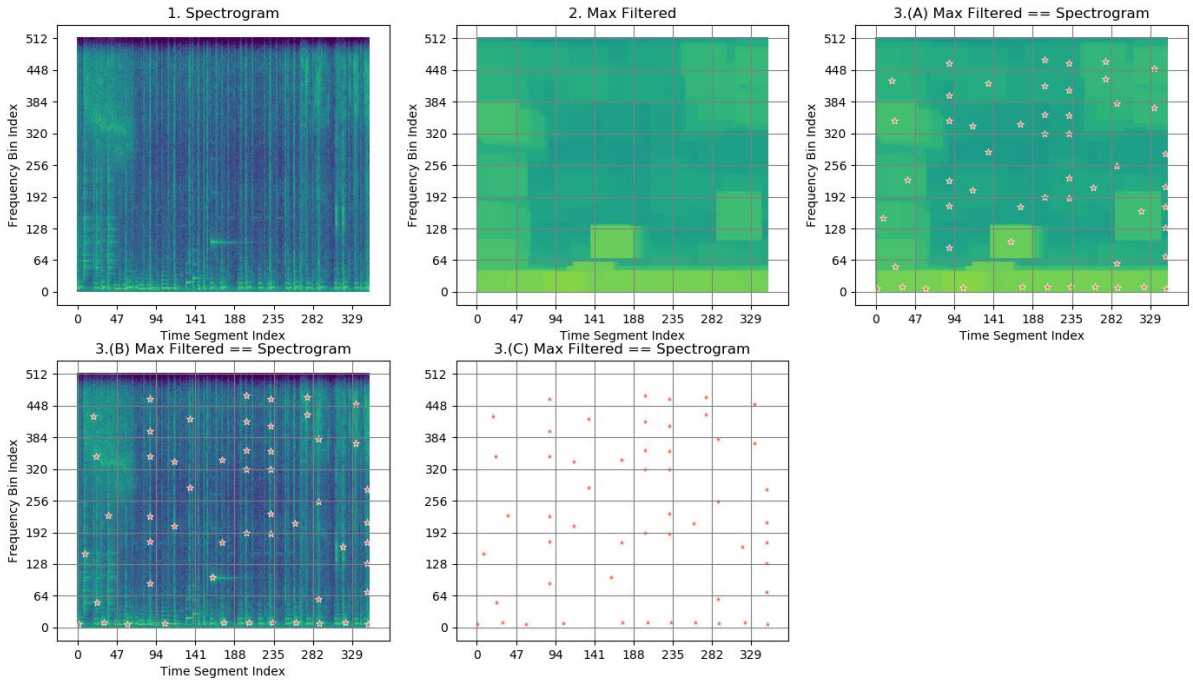

Figure 7-2: Using a 2D Maximum Filter for Spectrogram Peak Detection

The gridlines in Figure 7-2 show the filter size, which we set to 2 seconds by 500Hz. The original paper says that a spectrogram should have "reasonably uniform coverage." If we were to partition the spectrogram into perfect grids and find one maximum point in each subsection, that would lead to uniform coverage but we believe would result in a poor recognition rate. By using the maximum filter, we achieve "reasonably uniform coverage" that avoids this problem, while still allowing the user to change the density criterion by altering the filter size.

This process worked well for the initial set of data we were working with, but when we began building the database of 10,000 songs, the first song was generating thousands of more peaks than it should have been. We visualized the detected peaks and noticed there was an unusual amount evenly spaced in the last few seconds of the 6-minute song. The cause of this was that the song faded out into absolute digital silence, meaning the time-frequency bins were all set to 0. These were detected as peaks because the maximum filtered spectrogram and the original spectrogram were equal in all of these time-frequency

bins. This could also be an issue at the top frequencies of a spectrogram if the recording was sampled at a lower sample rate than the analysis occurred at, and thus the high frequency bins would all be 0. To solve this, we simply added Boolean logic that the detected bins could not have a magnitude of 0.

Figure 7-4: False Positive Peaks Due to a Fade Out

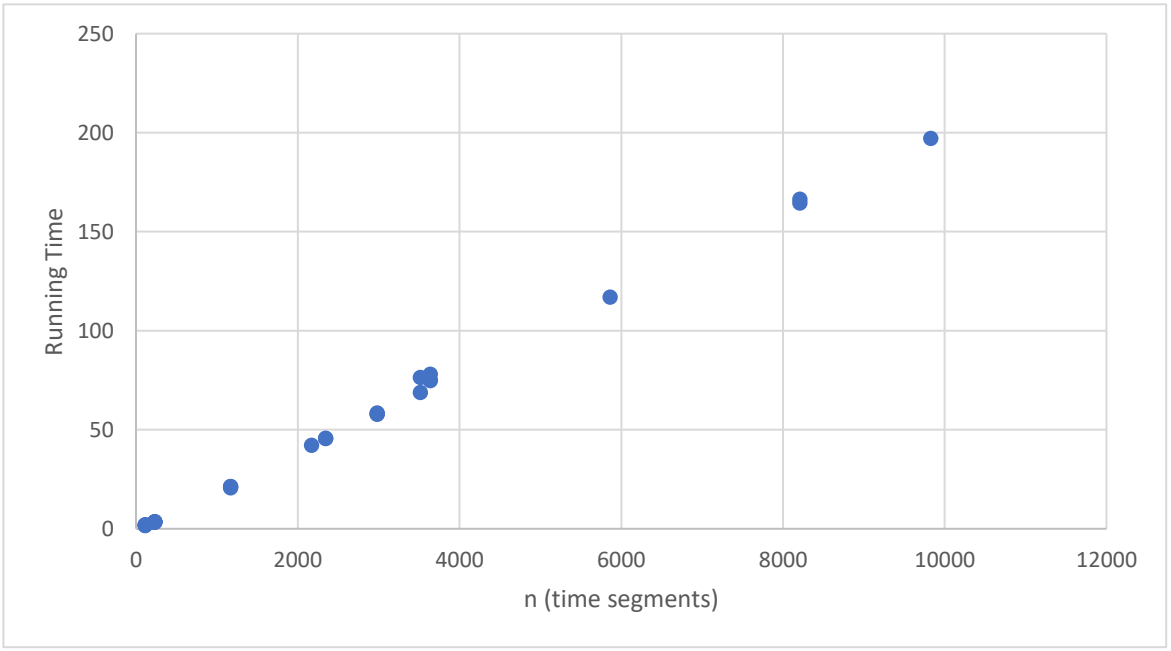

Figure 7-3: Running Time of FindSpectrogramPeaks()

We measured this process with a spectrogram of 513 frequency bins and n time segments from 115 to 9287 (or 20 to 450 seconds) and averaged the results over 100 repeats. The results are shown in Figure 7-3 and suggest that it is $O(n)$.

7.4 Histogram Clustered Peak Detection

Wang does not elaborate how they do peak detection on the songs' fingerprint-offset histograms except, "This may be done by sorting the set of δt_k [time-offset delta] values and quickly scanning for a cluster of values." In our implementation, we created a histogram that had a bin width of exactly 1. Then we applied a 1d uniform filter of size 2 from SciPy's image processing package, before getting the maximum peak value from that. We kept track of the maximum across all the possible song histograms. By using the uniform filter, we prevent the following issue. A song's histogram may have 2 neighboring bins with a value of 25. Another candidate song's histogram may have one bin with a value of 26. That second candidate would be chosen as a match, even though the first candidate is more likely. By using the uniform filter of size 2, bins are averaged with their immediate neighbors. So in the example given, the array with a single 26 would have a filtered peak of 13, but the first array of 2 adjacent 25s would still have a filtered peak of 25 and would be correctly chosen as the matching song.

One might be tempted to simply pick the song with the greatest number of matching fingerprints but we found this does not work. For example, in Figure 6-5 the correctly matched song only had 83 matching fingerprints with the audio clip, while another candidate song had 503 matching fingerprints. The reason for this is that longer songs and songs with more dynamic activity, and thus peaks, will match more fingerprints. But those fingerprints will not necessarily cluster into a single time-offset delta bin.

7.5 Binary Searching for Fingerprints in Target Zone

We initially selected peaks in a target zone to combine with the anchor peak for generating fingerprint hash-keys by doing simple Boolean indexing on a Pandas Dataframe of all the spectrogram peaks. We wrote a second implementation that before iterating through all the anchor points, first created a sorted

copy of the time locations and created another sorted copy of the frequency locations. These copies contain the index of the original DataFrame. For every peak, we found the peaks within its target zone by doing binary search on both sorted copies. We then performed the Boolean conjunction of those results and used them to index the original DataFrame. In a song that was 1.3 hours (the longest in the database) and had 19,915 peaks, we found that the original method of finding peaks in a target zone of 6 seconds by 1400 Hz took 1.65ms of wall clock time (averaged from 100 repeats) and the binary search method took 0.92ms. On a 15 second clip with 73 peaks, we found the original method took 1.5ms and the binary search method took 0.78ms. We could probably speed this up further with a sweep line algorithm but given that the input size is not going to arbitrarily scale up, that seems like an unnecessary optimization.

7.6 Multiprocessing and Multithreading

The largest bottleneck to the pipeline was loading the audio data due to the resampling process which took about 50% of code execution time. We do not think that resampling falls under the scope of this algorithm, so we did not seek to improve that. However, we did minimize its impact by employing multiprocessing and multithreading.

For building the database, we divided the list of mp3s to add and used the python multiprocessing library to spawn an arbitrary number of processes to work in parallel. Because we were using a Mongo Database, we did not have to worry about shared memory and this modification was relatively painless. The final 1000 songs were inserted into the database in 25 minutes using this method, instead of several hours. As an aside, the use of MongoDB is not required in our implementation because we abstracted all the database methods and included a class to run it in RAM with Python dictionaries. However, due to time constraints we only rigorously tested the MongoDB version.

For performing the recognition rate tests, we used the python multithreading class. We used the thread safe Queue to load mp3s in parallel as the test was being performed. The test of 250 songs with 3 clip lengths and 11 SNRs took 1.2 hours instead of 6 or 7.

8 Conclusion and Results

8.1 Recognition Rate

The original paper published their test results with a database of 10,000 songs, so we chose to also create a database of 10,000 songs. Our database had a wide variety of genres, including Classical, Hip-hop, and World music. Most songs were Indie Rock. The 10,000 songs were from 832 different artists and 1,086 different albums. Like Wang, we measured the recognition rate of 250 randomly selected songs. A list of the songs can be found on our code repository. Each song was tested with three different clip lengths, 15,10, and 5 seconds, taken from the middle of the song. Each clip was tested with additive noise with 11 different signal-to-noise ratios from -15dB to 15dB.

We converted all the audio to mono before processing. We down-sampled the audio to be 8kHz, which means it could represent frequencies between 0Hz and 4kHz. The spectrograms were computed with 513

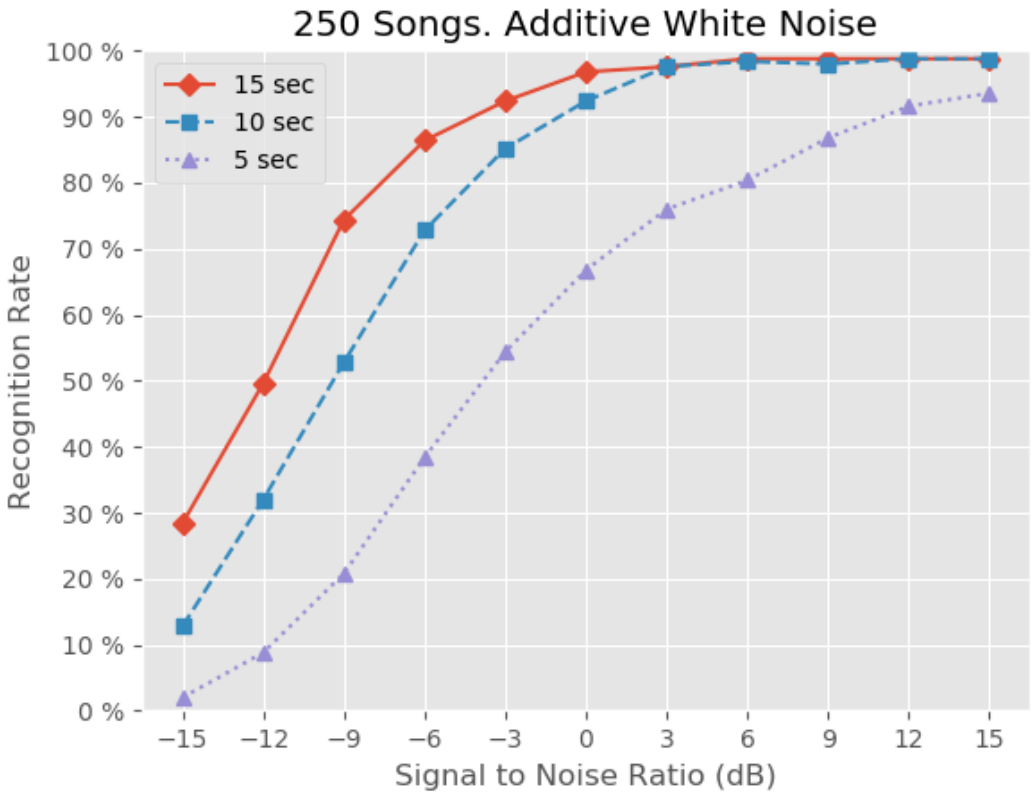

Figure 8-1: Recognition Rate with 4750 songs in the database –White Noise

frequency bins and with 1024 sample length time segments, overlapped by 683 samples. In other words, the temporal resolution was 0.043 seconds per time segment and the spectral resolution was 7.8125 Hz per frequency bin.

We initially generated white noise to use as the additive noise. We first tested the recognition rates when we had built the database halfway, with 4,750 songs and 24 million fingerprints. The results can be seen in Figure 8-1.

When we finished building the database of 10,000 songs and 49 million fingerprints, we retested the recognition rate. As can be seen in Figure 8-2, the rate worsens as expected, but not dramatically. Our 15 and 10 second curves are comparable to Wang, but our 5 second curve is noticeably worse. We suspect this is because our chosen density was smaller than theirs (they never specified) and therefore the 5 second clips did not have enough fingerprint matches to work with.

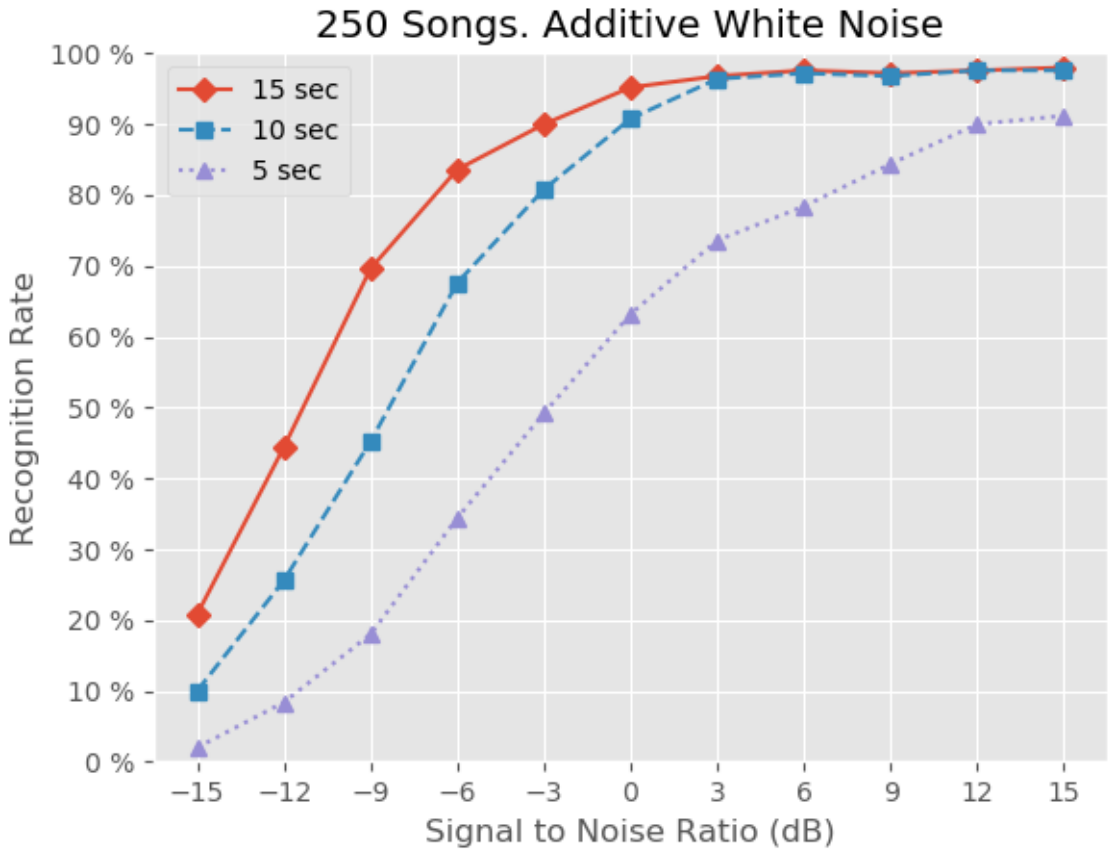

Figure 8-2: Recognition Rate with 10,000 songs in the database – White Noise

Instead of white noise, Wang used noise recorded from a crowded pub. We found a suitable recording of a crowded pub, which can be found in our code repository, and reran the test with that.

As can be seen in Figure 8-3, the performance worsens. For the 15 second clips, the SNR of 0dB drops from 95% to 92%. The SNR of -6dB drops from 83% to 63%. Wang claimed to achieve a performance of 85% at -6dB but they did not provide their noise audio. It is possible that their noise was less degrading than ours, perhaps because it may have had a narrower bandwidth. It is also possible that their target zone size and density criterion led to more fingerprints in their database.

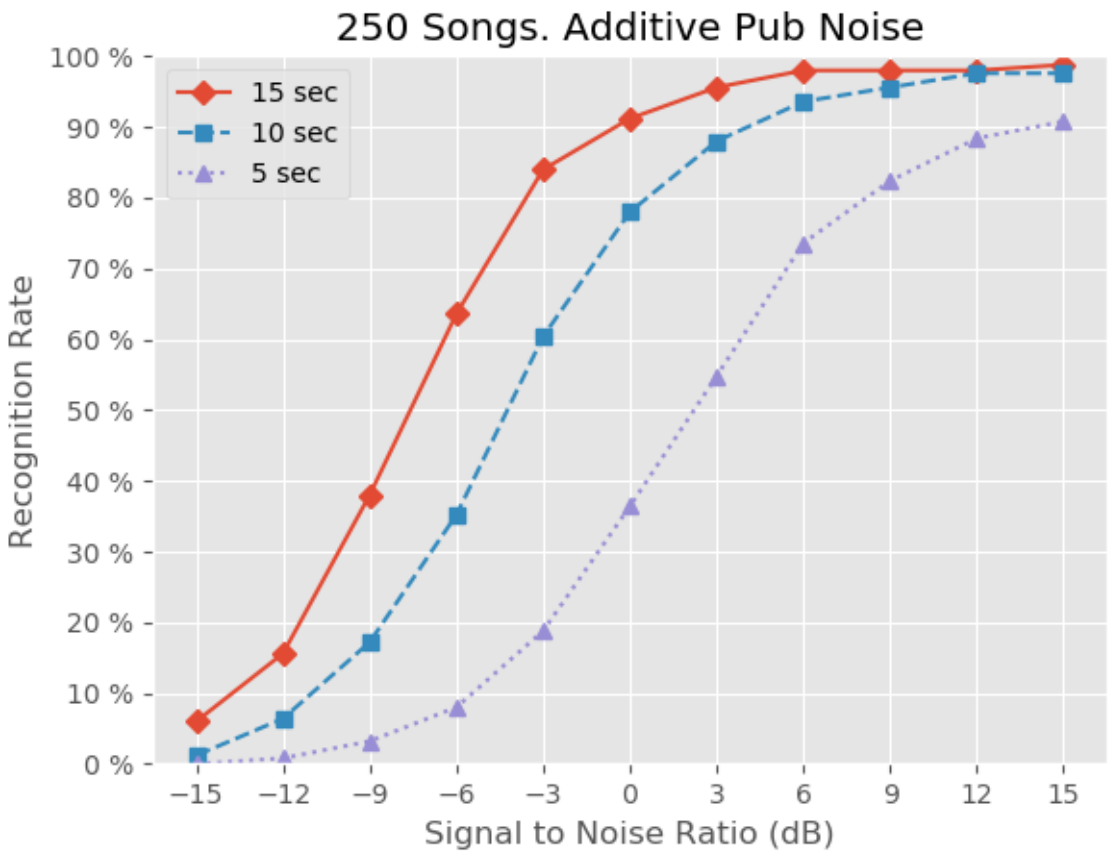

Figure 8-3: Recognition Rate with 10,000 songs in the database - Pub Noise

9 Appendix: Code Repository

GitHub: https://github.com/lukemcraig/AudioSearch

Detailed instruction for installation and use of the code can be found in the README.md in the repository.

10 References

[1] A. L.-c. Wang, "An industrial-strength audio search algorithm," in *Proceedings of the 4th International Conference on Music Information Retrieval*, 2003.