SHAZAM

INTRODUCTION

This paper outlines our approach and implementation of a variation of the popular application, Shazam. We detail our process of data collection, data preprocessing, data modeling, alignment matching, and experimentation. We also discuss our results and possible future extensions of this work. The essence of this work is to experiment with the generic Shazam algorithm of defining unique structures off of relatively short pieces of audio in a way that allows for efficient space use for storage, as well as efficient computation for performance. At a high level, we will be taking audio files, converting them into spectrograms, defining the peak points, and generating finger prints. While extrapolating complex data into relatively simplistic forms, we are still able to maintain consistent distinguishability between songs as well as similarity between the same song, even when background noise is added. We also stress tested our system not only against background noise, but limits of audio length, to show how little information is needed to accurately predict songs. This speaks greatly to both the novelty of the algorithm used, as well as the ability to use relatively simple processes to perform very useful tasks.

METHODS AND DESIGN SPECIFICATION

DATA COLLECTION AND PREPROCESSING

All of the songs we used for were collected from freemusicarchive.org. All songs from this website are free to use and download. For our project, we collected 10 random songs for 10 different genres in order to have a wide array of different styles of music, while still having multiple samples from the same genre to test against similarity.

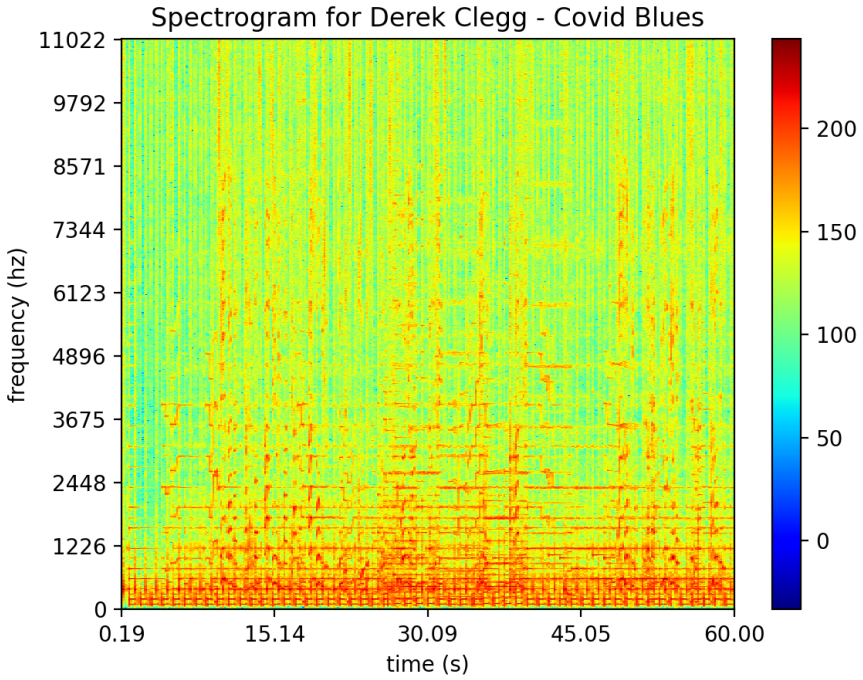
Our preprocessing pipeline is separated into two stages. First, we get the songs into a format tenable for use with the libraries and technologies we will be running. Specifically, we converted the song files which were in stereo mp3 format into mono wav files using AudioSegment from pydub. We then took the average of the two channels and wrote the result into a wav file. For the second part, we apply a bandpass filter to remove all frequencies below 10hZ and above 10kHz. From there, we downsample the data by a factor of 4, decreasing our sampling frequency from 44100Hz to 11025Hz.

FINGERPRINTING

To fingerprint a wav file, we first generate a spectrogram, select the points with highest amplitude, and finally generate hash values for the peak points and store them in our database.

SPECTROGRAM

We used the matplotlib.mlab.specgram to generate our spectrograms. We selected the default hanning window with a size of 4096, a sampling rate of 11025Hz, and an overlap ratio of 0.5. In each time window, Fourier Transform is repeatedly computed to obtain the frequencies, and amplitude. We then scale the frequency axis for the spectrogram, and convert from amplitude to decibel to finally something that looks like:

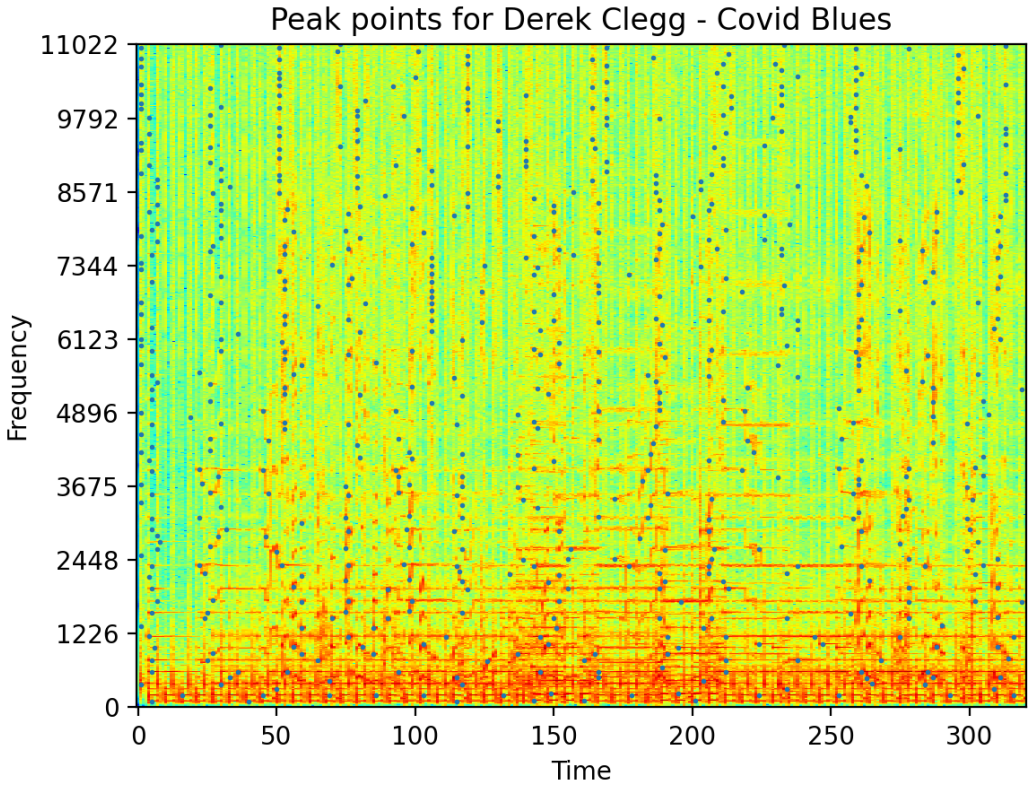


We interpret the spectrogram as the amplitude as a function of time and frequency. The regions in time and frequency which have the highest amplitude are colored in red. The weakest in blue.

FIND PEAKS

Peaks are regions in the spectrogram which have the highest amplitude. Note we cannot analyze points, as there would be too many peaks. We only want the regions where a certain frequency was the strongest. If we can find these peak regions, we can discretize the spectrogram into a set of integer points. There are numerous methods to extract these peaks. We will treat the spectrogram as an image and use image processing tools from the scipy library to find the regions with highest amplitude. We can create a square morphology mask that is parametrized by what

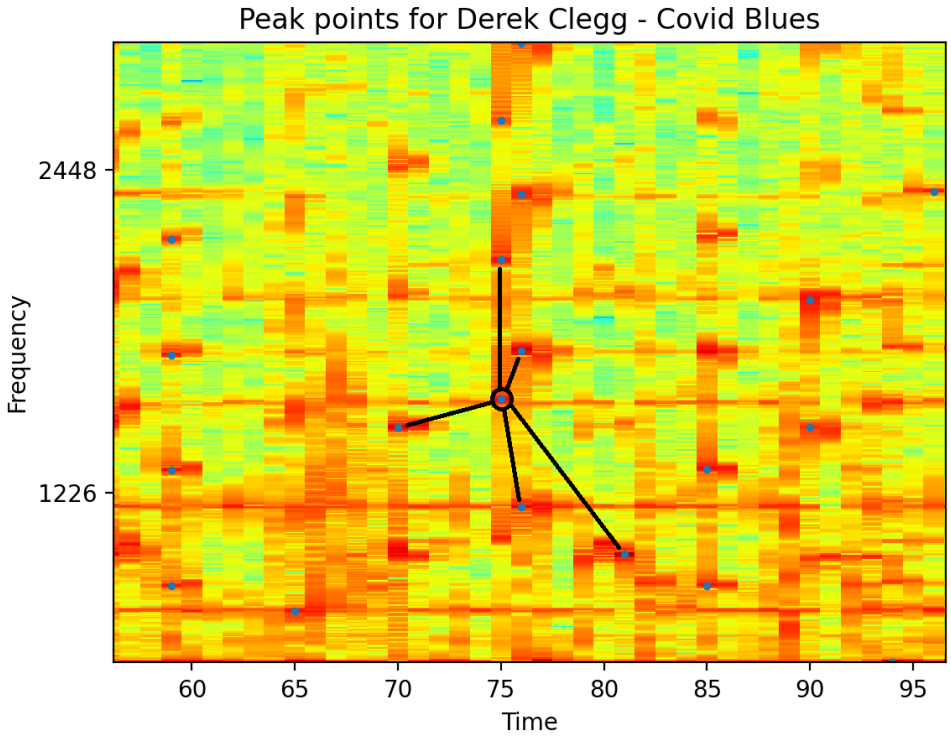
we call the neighbourhood size. This just represents the size of the region we consider when finding local maxima. We then find the local maxima using the mask using a simple scipy function. If the neighbourhood size is small, then we will have more peaks, and if the neighbourhood size is large, then we will have fewer peaks. The peaks for the song Covid Blues by Derek Clegg (same as diagram above) are:



We have set the peak neighbourhood size to 15, and the minimum amplitude for a region to be considered intense to 10dB. Increasing any of these parameters increases the number of peak points and decreasing them decreases the number of peaks. These time-frequency peak points can be used as distinguishers. For this project, we will hash the peak points.

Generate Hashes

We cannot simply hash the points, since different songs could have the same hash. We instead create a hash of pairs of peak points, and the relative time difference between them. So for each point, we generate fan value number of points. The fan value represents how many different neighbours we extend out to. The following is an example of generating hashes for a peak point with fan value equal to 5:



We use SHA256 as our hash function. Here | is concatenation. We only take the first 20 bits of the hash to save space. Doing so did not impact performance or correctness since the probability of collision in this small program is very small. Formally we define hashing as follows:

$$\text{For peak points } (x_1, y_1), (x_2, y_2), \qquad h = \text{SHA256}(y_1 | y_2 | t_2 - t_1)[:20], \;\; t_1$$

We can now store the hash value, time offset (from the beginning of the song) pair into our database and begin recognition!

DATABASE

Since we were working with a small dataset, we did not set up a SQL database. We simply created a database class in which we stored fingerprints. The database class has 2 methods, add fingerprint, and search. Add fingerprint is used to add a fingerprint object which holds fields hash, song id, and time offset. The search method is used for finding matching fingerprints in the database. More on this will be visited in the matching section.

If this work is to be extended, tools such as AWS to host SQL databases would be used which offer faster querying and many other benefits.

MATCHING

Once we have fingerprinted all the songs in our dataset, we cannot simply compare the fingerprint hash values of the recorded audio, with the fingerprinted hash values of the songs in our dataset. Doing so only work in the case that the beginning of the recording is also exactly the beginning of the corresponding song. That is, solely comparing hash values only works when the recording is aligned to the beginning of the fingerprinted song. Though this is rarely ever the case and so we need another method.

You might be wondering why we decided to store the absolute offset in our Fingerprints. This is exactly what we need to start matching. Let's call the full song's fingerprint's offset the absolute time offset. Let's call the recording's offset the relative time offset. Assuming that the playback speed of the recording, and the song are the same, we know that in the recording and in the full fingerprinted song, the distance between absolute and relative offsets are exactly the same. Note that this is a fair assumption to make, since most people play music at 1.0x speed. With this, we can now distinguish the true matches by storing the difference between the absolute and relative offsets as this will be the same at any point in time. Thus, for song recognition, simply find all matching hashes in the database, extract the absolute offset from the database for each match, and store the (song_id, offset difference). To predict, simply return the (song_id, offset difference) with the highest count.

EXPERIMENT SETUP

1. RECOGNITION ACCURACY AS A FUNCTION OF RECORDING DURATION

In this experiment, we want to determine how many seconds of recording audio does our app require to successfully recognize sufficient songs in our dataset. Thus, we record the recognition accuracy on our 100 songs, 10 songs from 10 different genres, when the recording is 1 seconds long, 2 seconds long, all the way up to 15 seconds long. We expect the recognition accuracy to be above 90% after 10 seconds. Regardless, accuracy should increase linearly with the recording duration.

2. NOISE RESISTANCE

Signal to noise ratio (SNR) is a metric used to measure how noisy a signal is. A high SNR, say 40, means that the signal is 40dB louder than the noise. A low SNR, say -40, means that the noise is 40dB louder than the signal.

We want to determine at what signal to noise ratio (SNR) does our system perform at below 80% of its original accuracy. For this experiment, we will fix the recording duration to 10 seconds, since that's what our system requires to achieve almost 100% accuracy. We will add Additive White Gaussian Noise (AWGN) to our audio files of desired SNR. To add noise at a certain SNR, we first compute the root mean square (RMS) of our signal, and determine the root mean square that we require for our noise. This is given by :

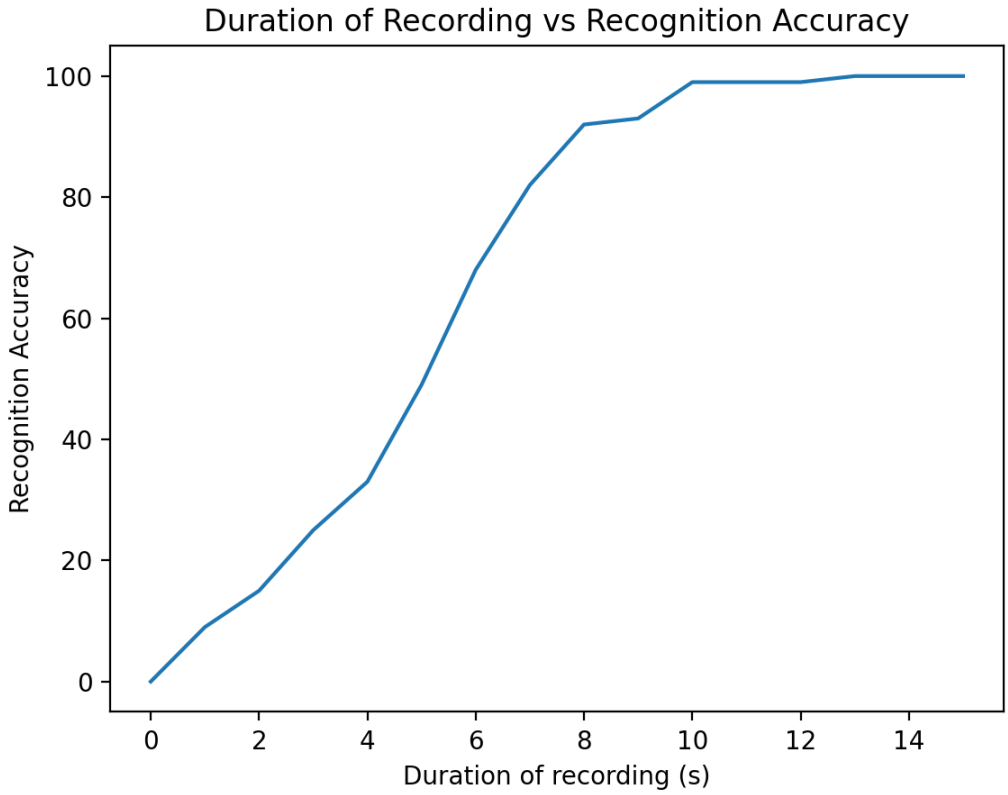
$$RMS_{noise} = \sqrt{\frac{RMS_{sig}^2}{10^{\frac{\text{SNR}}{10}}}}$$

Since this is white noise, our mean is very close to 0. And thus, the gaussian standard deviation is equal to the RMS of the noise signal. Thus, we can randomly sample noise from Gaussian(0, RMSnoise) using numpy in python. Finally, we add this noise to our original signal and evaluate accuracy. For this experiment, we will evaluate our system's performance for SNR 40, 37, …, 0, …, -37, -40. We expect that the system performance will continue to decrease as the signal to noise ratio decreases.

Results and Analysis

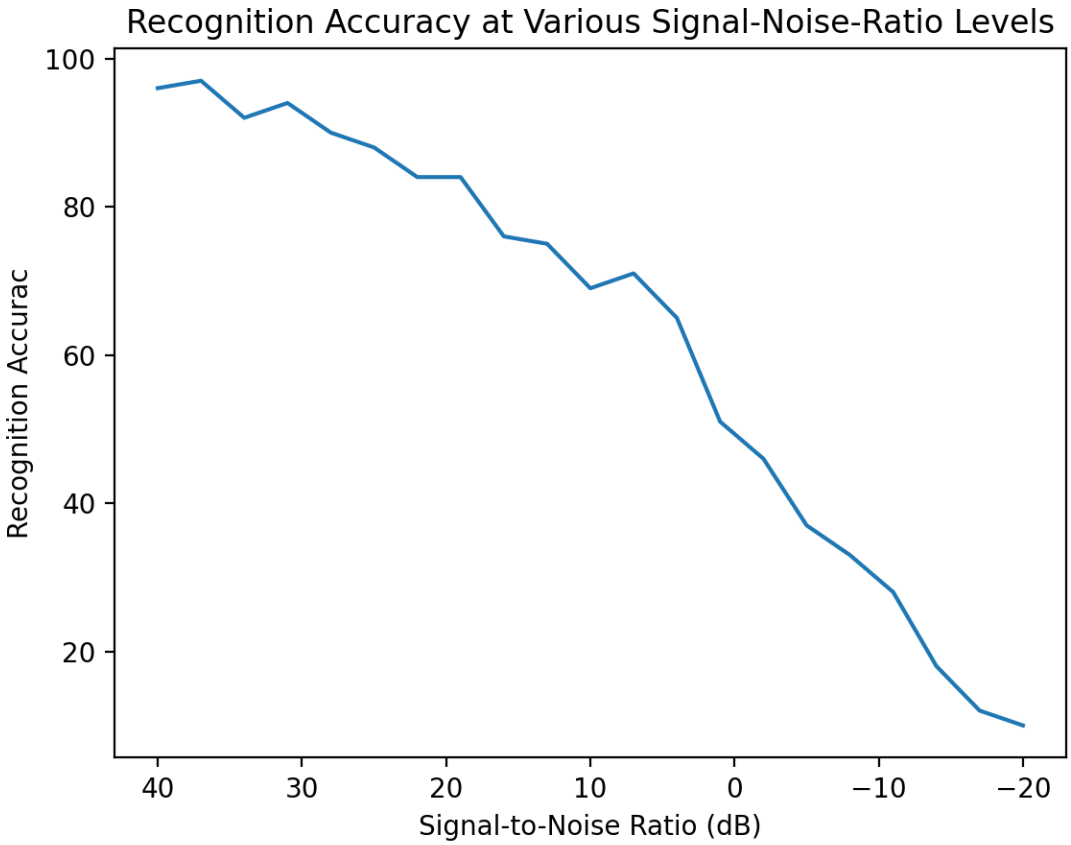
1. Recognition Accuracy as a Function of Recording Duration

As expected the recognition accuracy increased linearly with the recording duration. Our system performs very well, and successfully recognizes about 98/100 songs in our dataset after 10 seconds. In most real world applications, 10 seconds to recognize music is very reasonable. We are confident we can make improvements to this system by tuning some hyperparameters, but we will save that for future work. The system performs much better than expected.



2. Noise Resistance

As expected, the recognition accuracy decreases as more noise is added to our signal. We see that at about 22 SNR, the system performs below 80% accuracy. Therefore, if users of this system required 80% accuracy, then we can say that our system is resistant to 22 dB SNR which is pretty good.



At around 4 dB SNR, the system falls below 50% accuracy, and is unable to tell the noise from the signal.

Discussion and Future Work

Correctness

Overall, our system performs very well and has exceeded our expectations. We were able to recognize all of the songs in our dataset and so in terms of correctness, our system definitely passes.

Run-Time

Our system had a very high accuracy, and the time for matching is fairly quick. However, we have only ran this system on 100 songs. For each song we generate an average of 1132 fingerprints. Currently, the system uses a linear search to find potential matches in the database. This worked for our small dataset, but it does not scale well. Our system would be very slow on a large dataset.

SPACE-COMPLEXITY

Our system stores exactly 106458 fingerprints for the entire dataset. Each fingerprint holds a the first 20 bits of the SHA256 hash, a 32-bit integer representing the hash value, and assuming the average song name is 20 characters, a 32 bits * 20 = 640 bit string for the song id. So one fingerprint takes up at least 20+32+640 = 692 bits. Therefore, our system takes up 106458*692 bits, or ~86 megabytes. This is not bad considering our very vanilla Database design, however, it can be improved significantly. We can instead store the first 20 bits of the SHA256 hash of the song_id. This would save us (640-20)*106458 bits of memory which is about 8 megabytes of memory. This is from changing the way we store the song_id alone.

Moreover, a real database such as mySQL stores values much more efficiently, and does not require fingerprinting every time you want to run the script. You can have a SQL session running, and add fingerprints sequentially instead of adding them all at the beginning of an experiment.

Use a real database, like mySQL faster query time, and don't have to fingerprint every time you want to recognize could just have a session running

FUTURE WORK

We certainly exceeded our expectations for this project, but as we were working on it, saw a lot of room for improvement. For the purposes of extending this work, we will touch on various aspects of the project that could be augmented to allow for scaling and increased performance.

A low hanging fruit that would allow for better storage and querying would be to utilize a standard database, like mySQL or Postgres, and host it on AWS or GCP. Not only would this remove the need to use local RAM to store values in python classes [or PICKLE FILES], but it would also tremendously increase performance on adding and searching the database, as these types of queries are well optimized. Tools like ElasticSearch may also be useful in regard to performance.

Another nuance that we did not commit too much time to was the choosing of various hyper parameters defined across the codebase, including filter types, frequency cut offs, and downsampling rate, peak neighbourhood size, minimum amplitude, fan value, window size, overlap ratio, etc. Building out an online-machine learning model that is continuously tuning these parameters based on successful and failed matches could help improve accuracy in the long run.

For extension involving pushing this software into users hands would involve user trials to determine what the best trade off between storage and performance is, and whether having more or less fingerprints is worth it, in a way that a machine learning model couldn't, on its own, determine.

If this were to be extended into a full-fledged application, other questions would have to be considered. Primarily, the trade-off between storage and performance would become an important question worth investigating, as well as the trade-off between having an exhaustive database of songs that is slower, or a more targeted and smaller database that has better performance. Questions like these would most likely require more than a machine learning model to answer, and more product-oriented decision making.

One further experiment we would like to conduct is one to determine the impact of the number of fingerprints on performance. Currently our database holds 106458 fingerprints for 100 songs. However, there are numerous parameters which can increase or decrease the number of points on a spectrogram we consider peaks, and hence increase the number of fingerprints for our dataset. When there are more fingerprints in our database, we noticed that we achieved higher accuracy with lower duration of recordings. However, the matching took longer since there were more fingerprints to compare with. Hence a performance trade-off with space and run-time. We want to conduct an experiment where we incrementally increase the number of fingerprints and concurrently measure matching time and accuracy. We would hope this would help us estimate the ideal number of fingerprints that gets us the best memory usage without sacrificing run-time.

All in all, we constructed this project in a way we believed best expressed the algorithms and the technologies involved. However, lots of additional challenges could be tackled once the scope and scale of this work is increased. Here we've outlined only a few of the many possibilities!

REFLECTION

Every aspect of this project was helpful in either reinforcing an existing concept of computational audio, or helping us learn about a new one.

Within the data processing pipeline, we became even more familiar with the various audio file types (mp3, stereo wav, mono wave, etc) and how best to interact with them in a python environment. Through lots of trial and error, sound testing, and data visualization, we created a pipeline generalizable across our database that ensured sound quality and reduced file corruption. We also got to try out downsampling (a concept only recently detailed in the course notes) and observe the performance rewards associated with simplifying the underlying data. These steps, along with other preprocessing steps like Fourier transforming and hanning windowing, were helpful in reinforcing various concepts taught during lectures.

The process of fingerprinting songs was extremely interesting in both its ability to map a complex song into a relatively simple structure, as well as its ability to maintain consistent values during noise disruption.

All in all, reproducing an application that we have both used frequently would have been enjoyable and fulfilling on its own. But having now understood exactly how and why Shazam works the way it does has helped us further develop our understanding of various computational audio concepts and encouraged us to continue diving into both this application and the space in general.

Sources Consulted

http://coding-geek.com/how-shazam-works/

https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf

https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0

https://willdrevo.com/fingerprinting-and-audio-recognition-with-python/

https://medium.com/analytics-vidhya/adding-noise-to-audio-clips-5d8cee24ccb8

https://github.com/itspoma/audio-fingerprint-identifying-python