Music Identification through Audio Fingerprinting

Group 1

Yashraj Kakkad - *AU1841036*
Prayag Savsani - *AU1841035*
Yashil Depani - *AU1841005*
Harshil Mehta - *AU1841010*

Guided by - Prof. Ashok Ranade
Signals and Systems, Monsoon 2019

I. Objective

The objective of the project is to recognize a song from an arbitrary 30 second sample through appropriate audio fingerprinting techniques. To accomplish that, a number of songs are broken into chunks. The best frequencies across a spectrum are taken and they are hashed in an appropriate data structure. The obtained input from the user undergoes a similar procedure and is then compared with the hash values of the song. The song which matches with the input the most is displayed as a result to the user.

This was a very brief outline. The coming sections exactly explain what we have done.

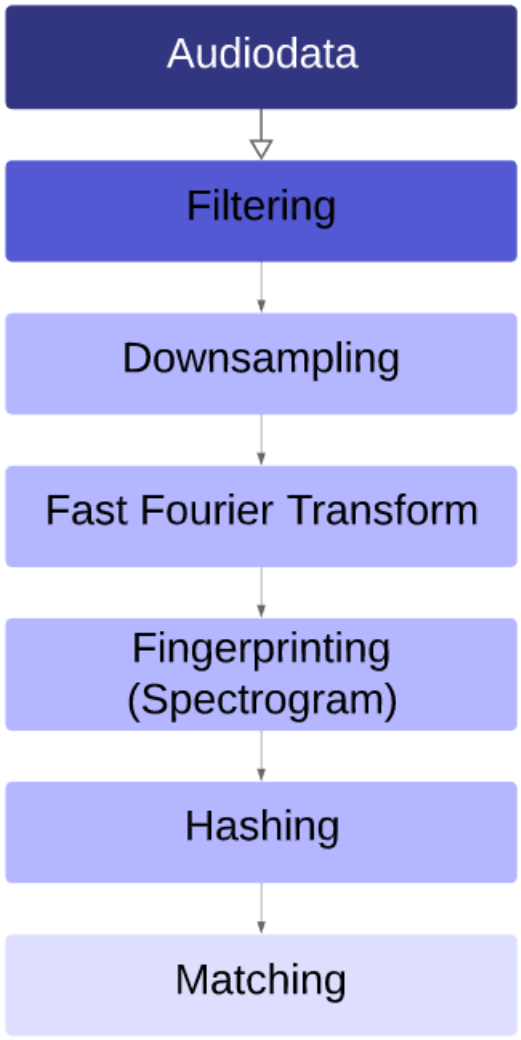

Fig. 1: Flowchart explaining our fingerprinting algorithm in brief

II. Sampling

Music is typically sampled at 44.1 kHz. This is because of a theorem by Nyquist and Shannon which requires $f_d >= 2f_s$ where f_d is the digital frequency and f_s is the sampling rate. The number, in practice, is greater than 2. Maximum sound frequency, as we know, is 20 kHz.

We'll be using Fast Fourier Transform at a later stage to obtain the frequency spectrum of several chunks of the song. Running it on even a few hundred songs would take days. Therefore, we downsample the recorded sample by a factor of 4. As a consequence, the maximum sound frequency in our audio sample comes down to 5kHz. This is not a problem as the frequency spectrum of most of the song lies below 5kHz. The gains are therefore much higher than the losses.

III. Filtering

Aliasing refers to the distortion that results when a signal reconstructed from samples is different from the original continuous signal. We filter the high-end frequences before downsampling to avoid this phenomenon. We use a 5th order butterworth low-pass filter to achieve the same.

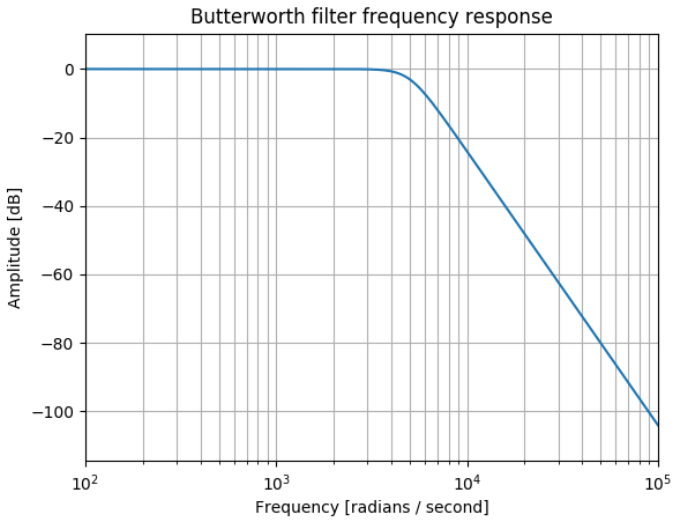

Fig. 2: Frequency response of a butterworth low-pass filter from Sci-py package in Python

IV. Fast Fourier Transform

Discrete Fourier Transform converts a signal from the time domain to the frequency domain. The

formula is:

$$X(n) = \sum_{k=0}^{N-1} x[k]e^{-j(2\pi kn/N)}$$

Here,

- N is the window size
- X(n) is the n^{th} bin of frequencies
- x(k) is the k^{th} sample of audio signal

Fast Fourier Transform is an algorithm that computes Discrete Fourier Transform. As the name says, it is "faster" than actual Discrete Fourier Transform. To make a comparison, Discrete Fourier Transform requires $\mathcal{O}(N^2)$ while its counterpart only requires $\mathcal{O}(NlogN)$, where N is the number of samples. This is a huge improvement since N is typically quite large in our case.

Speed is important in this case as the quicker we get the answer, the better experience we provide to the user. Also, the database of songs to be maintained is large.

V. Windowing

We cannot directly perform Fast Fourier Transform on the whole song. A typical song has many frequencies and the input from the user can be any part of the song containing any of these frequencies.

We therefore intend to apply Fast Fourier Transform on small chunks successively. For doing so, we need to extract the appropriate parts of the song. We do that by multiplying our audio signal with an appropriate window function. But by "multiplying", we are introducing new frequencies that did not exist before in the audio signal. This is known as spectral leakage.

Window size is 4096 samples. That makes our spectral resolution to be 0.37 seconds, effectively meaning that the algorithm can detect changes in the song for every 0.37 seconds.

We cannot avoid spectral leakage in this case but we can handle its behavior by appropriately choosing our window function. Some window functions, like the "usual" rectangular window function, lead to spectral leakage. Others, like Blackman are particularly bad with noise. We choose Hamming Window for our purpose since its performance lies between the two extrema.

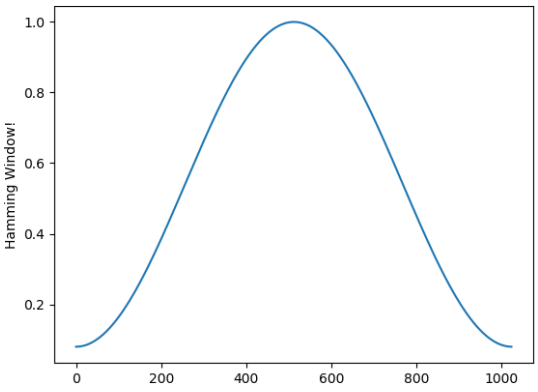

Fig. 3: Hamming Window Function

VI. Fingerprinting

An audio fingerprint can be considered a digital summary which helps to identify an audio sample from a given audio database. Audio fingerprints differ from standard computer fingerprints like SSHA or MD5 because two different files containing the same music must possess the same audio fingerprint which wouldn't be the case with their SSHA or MD5 fingerprints. This is why, audio fingerprints are generated using filtered spectrograms of audio samples which only contain data regarding the frequencies present in the song.

A. *Spectrogram*

A three dimensional graph where

- X-axis indicates time
- Y-axis indicates frequency
- Color indicates amplitude of a frequency at a given time

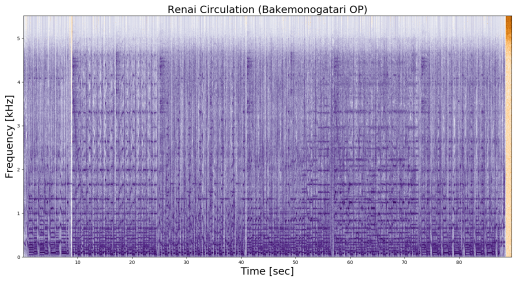

Fig. 4: Spectrogram of a full length song

Using the frequencies obtained from FFT, a spectrogram can be generated. For a filtered spectrogram,

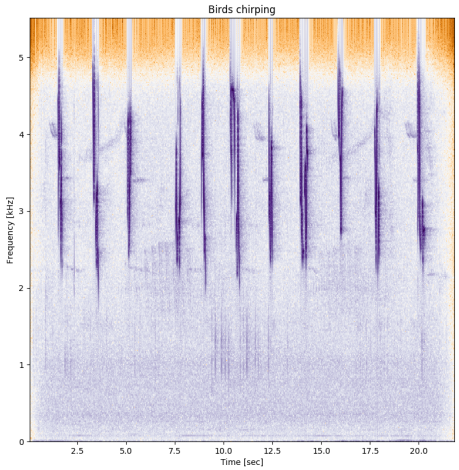

Fig. 5: Spectrogram of a song with sharp peaks

you need to extract the minimum number of frequencies that you need to uniquely identify a given audio sample.

B. *Filtering the ocean of frequencies*

While looking for the most important frequencies, we are looking for frequencies with the strongest amplitudes which are also called peaks. This is done because the algorithm needs to be as noise tolerant as possible. As different songs will have these frequencies in different frequency ranges, we divide our whole frequency spectrum (0-5 kHz) into 5 frequency ranges. They are -

- 30-40 Hz
- 40-80 Hz
- 80-120 Hz
- 120-180 Hz
- 180-600 Hz

Using this frequency data, we can now a plot the filtered spectrogram of the audio sample which looks like this - This filtered data effectively acts as our audio fingerprint and can now be hashed to decrease processing times.

VII. Hashing

Hashing is an efficient way to store these values as comparing hash values is an $\mathcal{O}(1)$ operation. Now that we have the most relevant frequencies across five bins for every window, they can be hashed into a single number and stored in the database. The hashing formula involves a factor for error correction conveniently called 'fuzzy' factor. This is there because recording conditions are not always perfect and there needs to be a variable factor to adjust these hash values. We use the following algorithm to hash our values: HASHING ALGORITHM GOES HERE

VIII. Matching

This is perhaps the most important part of the process. We perform the exact same operations on our recorded sample from the user. Should just matching the hash values help us identify which song it is? Maybe yes, but there's a more effective algorithm to accomplish the same.

Besides the hash values, we also have the relative time in which they're present in the song as the stored values are sequential. If:

- sample #1 of our recording matches with sample #25 of a particular song
- sample #3 matches with sample #28 and
- sample #8 matches with sample #32,

we can safely claim that the song recorded is the song we're talking about. This is because the relative offset in these cases is the same: $24 = 25 - 1 = 28 - 3 = 32 - 8$.

Song Name / Relative Offset				
Ariana Grande- 7 rings	2	5	1	6
Calboy- Envy Me	12	10	16	5
Charlie Puth- We Don't Talk Anymore	13	27	172	34
Billie Eilish-bad guy	3	9	7	76
Chris Brown	9	1	18	33
DaBaby- Suge	77	15	11	11
Ed Sheeran-Beautiful people	23	20	10	2
J. Cole- Middle Child	8	6	26	9
Dua Lipa- New Rules	7	13	45	8
Coldplay	0	29	50	20
City Girls	8	7	2	8
Halsey-Nightmare	21	17	41	18
Jonas Brothers-Sucker	31	78	44	22

Fig. 6: Illustration of the result of our matching algorithm. The song corresponding to the maximum offset is our answer.

We can summarize our matching process as under:

- Record a sample and perform the above steps on the recorded sample.
- Load the database of hash values

- For every hash value, identify the songs having that hash value. Keep track of the relative offset.
- Return the song (or few best songs) having a hash value with the maximum number of offsets across all songs and all hash values.

IX. Results

We tested the algorithm in several situations and here are the conclusions: We took random 30-second cuttings of the songs and put them to test. The success rate was 100%. The algorithm didn't fail even once.

However, in real world situations, several factors come to play. The quality of speakers as well as the microphone matters and there's some noise too. With a decent phone microphone and minimal noise, the accuracy is reasonably good.

Here's a comparison of the spectrograms of the same song and the same duration. The first one is a recording and the second one is cut from the song. As it can be seen, they are reasonably similar.

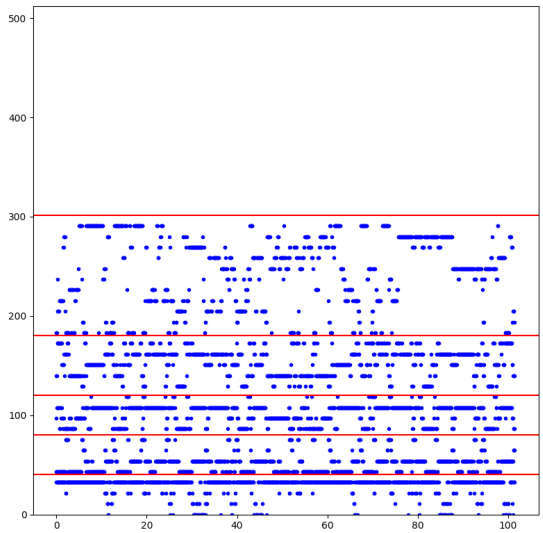

Fig. 7: Spectrogram of an *extracted* 10 second sample

We also tinkered other constants in our algorithm, like the samples per window, and settled with 4096 samples. We also took chunks of relative offsets by dividing them by a factor like 5 or 10, but that did not help much at the end.

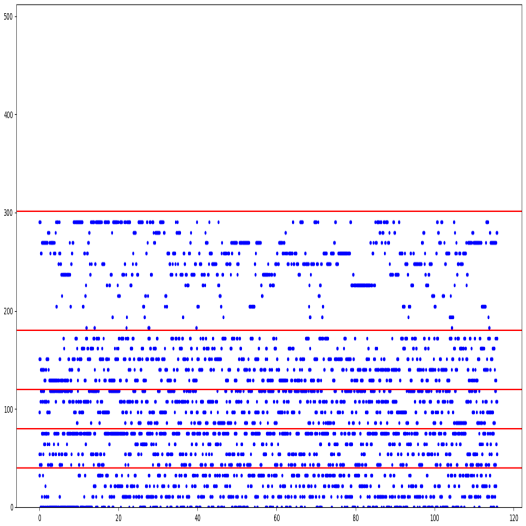

Fig. 8: Spectrogram of a *recorded* 10 second sample

pythonsong.py