A project report on

MUSIC SIMILARITY CHECKER USING FEATURE EXTRACTION

Submitted in partial fulfilment for the award of the degree of

Bachelor of Technology

in

INFORMATION TECHNOLOGY

by

ARINJAY JAIN

16BIT0059

Under the Guidance of

Dr. Mangayarkarasi R

School of Information Technology and Engineering (SITE)

MAY,2020

DECLARATION

I hereby declare that the thesis entitled "Music Similarity Checker Using Feature Extraction" submitted by me, for the award of the degree of *Bachelor of Technology in Information Technology to VIT* is a record of bonafide work carried out by me under the supervision of Mangayarkarasi R.

I further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place : Vellore

Date : 10 May 2020

Signature of the Candidate

CERTIFICATE

This is to certify that the thesis entitled "Music Similarity Checker Using Feature Extraction" submitted by **Arinjay Jain, 16BIT0059, School of Information Technology and Engineering**, VIT, for the award of the degree of *Bachelor of Technology in Information Technology*, is a record of bonafide work carried out by him / her under my supervision during the period, 01. 12. 2018 to 30.04.2019, as per the VIT code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The thesis fulfills the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place :Vellore

Date : 14 May 2020

Signature of the Guide

Internal Examiner

External Examiner

Head of the Programme

ACKNOWLEDGEMENTS

It is my pleasure to express with deep sense of gratitude to Mangayarkarasi R, Associate Professor, School of Information Technology and Engineering, Vellore Institute of Technology, for her constant guidance, continual encouragement, understanding; more than all, he taught me patience in my endeavor. My association with her is not confined to academics only, but it is a great opportunity on my part of work with an intellectual and expert in the field of signal analysis.

I would like to express my gratitude to <Chancellor>, <VPs>, <VC>, <PRO-VC>, and <Dean Name>, <School Name>, for providing with an environment to work in and for his inspiration during the tenure of the course.

In jubilant mood I express ingeniously my whole-hearted thanks to <Program char-name>. <Program Chair and designation>, all teaching staff and members working as limbs of our university for their not-self-centered enthusiasm coupled with timely encouragements showered on me with zeal, which prompted the acquirement of the requisite knowledge to finalize my course study successfully. I would like to thank my parents for their support.

It is indeed a pleasure to thank my friends who persuaded and encouraged me to take up and complete this task. At last but not least, I express my gratitude and appreciation to all those who have helped me directly or indirectly toward the successful completion of this project.

Place: Vellore
Date: 14 May 2020

Arinjay Jain

Executive Summary

In the growing world of music and ease of sharing of music, it is important for new artists to know if their song is similar to a pre-existing melody – thus allowing them to avoid being taken down on social media websites for infringing on copyright of the original artist.

Sometimes people might be watching a video and hear a song playing in the background without knowing the artist or the name of the song, this application will allow them to play the song into the mic for 10 seconds and search the fingerprint database to give them the name of the song (if present in the database). This will work even with someone speaking over the song while its being recorded to a certain degree as the hash generated for fingerprinting of the songs is robust.

The accuracy for original songs being matched in the database is quite high, but the accuracy drops when a cover of the same song is being searched for in the database. This is in contrast to other methods earlier tried where I extracted features such as spectral centroids, spectral bandwidths and Mel-frequency cepstral coefficients and tried to use them to match audio. This was a very slow and inefficient method.

CONTENTS

Page No.

List of Figures

(In the chapters, figure caption should come below the figure and table caption should come above the table. Figure and table captions should be of font size 10.)

List of Tables

List of Abbreviations

MFCCs	Mel-frequency cepstral coefficients
MFC	Mel-frequency cepstral
FFT	Fast Fourier Transforms
STFT	Short Time Fourier Transform
DFT	Discrete Fourier Transform

Symbols and Notations

The following notation is for DFT

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

1. INTRODUCTION

1.1. OBJECTIVE

The main objective of this project is to find a efficient way of extracting some features from audio files of published music and store them in a database. The user then can play an unidentified audio sample (minimum 5 seconds) while the program listens to it, extracts the feature from it and then matches them to the database.

The output would be the name of the song it matched from the database after comparing feature as well as the confidence level it has of finding a correct match. There are cases where we get false positives in some features, thus it is important to use the feature which is most accurate.

1.2. MOTIVATION

This was a personal problem I had faced for a long time, identifying songs from various sources without any prior knowledge. For example, while watching a movie a song starts playing in a scene in the background that I cannot identify on my own, I would usually resort to spending a lot of time searching online for that specific movie scene and hope to fin the name of the song and the artist that was playing at that time. While doing this, I came across the application called “Shazam” which did precisely this, it heard an audio clip for 10 seconds and ran it against its database with millions of songs to give a quick response – with a high accuracy rate – stating the song name, album and artist details.

This intrigued me a lot and I decided to create a simpler version using feature extraction from .mp3 files and storing the features in a database of my own. Then I could hear the input audio and extract features from it to match against the database and get a match.

1.3. BACKGROUND

Table 1: Literature Survey

SNo	Author/Date	Topic	Concept	Context	Findings
1	Hui Li Tan, Yongwei Zhu, Lekha Chaisorn, Haibin Huang (2009)	Sequential Rhythmic Information Retrieval for Audio Similarity Matching	It is a study on the rhythmic structure patterns of music compositions at different salience levels. They illustrate a sequential rhythmic information retrieval approach, based on the salience of detected onsets	Institute for Infocomm Research, A*STAR	The system proposed by the paper follows a 4 step process: it starts from Percussive Onset Detection, Onset prominence extraction, Onset prominence ranking and then Rhythmic Structure retrieval to gain information about the rhythm.
2	YA-DONG WU, YANG LI, BAO-LONG LIU (2003)	A New Method for Approximate Melody Matching	Presents a Linear Alignment Matching (LAM) method. It takes a geometrical approach and compares melodies by matching their pitch-time contours. A query method	Department of Computer Science & Engineering, Shanghai Jiaotong University	The system follows a 5 step process: It starts with a “humming” input as a query to be matched. The melody is extracted from the input and Note Sequence is formed. The sequence is passed through the LAM and a

			built on LAM has a high success rate of 90.3% for a database with 3864 songs.		match is made from the Melody Database. Following, a rank list is made.
3	Sazia Parvin, Jong Sou Park (2007)	An Efficient Music Retrieval Using Noise Cancellation	It is a study which extracts melody from music after applying noise cancellation to the music. This approach of creating a noise free signal at the start has a 94% retrieval accuracy	Korea Aerospace University Network Security Laboratory	The system proposed has a basic 2 part process but the sub processing in it is complex. In the noise cancellation step Perceptual Filter Estimation and Filtering is performed while in music retrieval step, smoothing, feature extraction and matching is performed with the music in the database.
4	Zanchun Gao, Yuting Liu, Yanjun Jiang (2015)	An effective method on content based music feature extraction	Music feature extraction methods involving frequency domain and time domain signal	Beijing University of Posts and Tele-communications	The proposed system uses linear regression and linear support vector machine models and achieves a very

			processing, wavelet analysis and singular value decomposition are mentioned in this paper.		high precision rate of 95.33%. It distinguishes music using various features such as Frequency feature, Auditory Perceptual Feature, and Statistical Characteristic of Beat.
5	Takahiro Hayashi, Nobuaki Ishii, Masato Yamaguchi	Fast Music Information Retrieval with Indirect Matching	In the approach, a small number of music clips called representative queries, which are randomly selected from a database, are used for fast computation	Department of Information Engineering, Faculty of Engineering, Niigata University	In the offline process, the similarities of each music clip in the database to the representative queries are recorded as a similarity table. In the online phase, the similarity between the actual query (the music clip given by a user) and each music clip in the database is quickly estimated by referring the

					similarity table.
6	Wei Li, Xiu Zhang and Zhurong Wang	Music content authentication based on beat segmentation and fuzzy classification	Previous audio authentication algorithms are mainly focused on either human speech or general audio with music as part of the test data, while special research on music authentication has been somewhat neglected. This paper proposes a new algorithm to protect the integrity of music signals.	NA	It is a three part process which consists of segmentation of music into beat based frames (this addresses the synchronization problem). Next robust hashes are generated from chroma-based mid-level audio feature which can appropriately characterize the music content and integrated with an encryption procedure to ensure the security against malicious block-wise vector quantization attack. Finally, fuzzy logic is adopted to make the authentication decision in the

					light of three measures defined on bit errors, coinciding with the inherent blurred nature of authentication
7	Frank Kurth ; Meinard Muller (2008)	Efficient Index-Based Audio Matching	They address a higher level retrieval problem, which is referred to as audio matching: given a short query audio clip, the goal is to automatically retrieve all excerpts from all recordings within the database that musically correspond to the query.	IEEE Member	In their matching scenario, opposed to classical audio identification, they allow semantically motivated variations as they typically occur in different interpretations of a piece of music. To this end, this paper presents an efficient and robust audio matching procedure that works even in the presence of significant variations, such as nonlinear

					temporal, dynamical, and spectral deviations, where existing algorithms for audio identification would fail.
8	Weijiang Feng ; Naiyang Guan ; Zhigang Luo (2016)	High-performance audio matching with features learned by convolutional deep belief network	They train the Convolutional Deep Belief Network using the TIMIT dataset and implement it through MATLAB. They have different covers of the song in the dataset for higher matching accuracy.	Institute of Software, College of Computer National University of Defense Technology, Changsha, Hunan, P.R. China	In this paper, they propose to utilize the features learned by Convolutional Deep Belief Network (CDBN) to enhance the performance of audio matching. Benefit from the strong generalization ability of CDBN, our method works better than CENS Chroma Energy Normalized Statistics based methods on most audio datasets. Since

					the features learned by CDBN are binary-valued, we can develop a more efficient audio matching algorithm by taking the advantage of this property.
9	E. R. Swedia, A. B. Mutiara, M. Subali and Ernastuti (2018)	Deep Learning Long-Short Term Memory (LSTM) for Indonesian Speech Digit Recognition using LPC and MFCC Feature	They are using Deep Learning Long-Short Term Memory (LSTM). The LPC (Linear Predictive Coding) and MFCC (Mel-Frequency Cepstrum) feature extraction was used as an input on the LSTM model and the level of recognition accuracy was compared. The LPC feature extract speech feature based on a	Gunadarma University, STT Cendekia	They used 7990 speech digits consisted of 12 LPC coefficients and 12 MFCC coefficients as training data, while 790 data was used to classify on LSTM that had been trained. The results show that using LSTM for recognize Indonesian speech digit, the MFCC feature extraction gets better accuracy result of 96.58%

			pitch or fundamental frequency, while MFCC extract speech feature based on the sound spectrum.		compared to the LPC feature extraction which amounts to 93.79 %.
10	Wang, Avery. (2003).	An Industrial Strength Audio Search Algorithm.	Extract and create hash tokens of songs and store it in database. The sample audio is also processed the same way and then matched.	USA: 2925 Ross Road Palo Alto, CA 94303	Their implementation followed a core of 3 steps: 1. Robust Constellations 2. Fast Combinatorial Hashing 3. Searching and Scoring. For heavily corrupted input of audio signals their time for each query was approximately few 100s of milliseconds. For a perfect audio input, they had less than 10ms recorded for each query.

2. PROJECT DESCRIPTION AND GOALS

The final goal of the project is to find a match between the input sample audio and the songs stored in the database using features extracted in an efficient way with keeping memory of database and speed of matching in mind. For this we are using ‘fingerprints’ of songs that are stored in a SQLite database.

The music similarity system consists of:

- Feature extraction capabilities
- Matching of fingerprints of audio input from mic to find a match
- Database to store songs and their fingerprint

There are various functions to achieve above tasks. The feature_extraction.py file uses the Librosa python library to extract features such as MFCCs and Spectral centroid and more and store them in a JSON format. We also plot these features on a graph using MatPlot library.

The collect-fingerprints-of-songs.py file takes all the songs in the mp3 file and fingerprints them in both channels – mono and stereo – and stores the hash values in the SQLite database.

The SQLite database has 2 tables – songs and fingerprints. Fingerprint table stores the song is, the hash and the offset, while the song table stores the song id and the name of the song.

When we run the recognize-from-microphone.py file with an argument for how long we want to listen (in seconds) the audio file starts recording for the specified time. The audio clip recorded is then fingerprinted and matched with the database to get the name of the song as output, along with the confidence level.

3. TECHNICAL SPECIFICATIONS

HARDWARE:

1. CPU – Intel i7 8750h
2. GPU – Nvidia GeForce GTX 1060
3. 16 GB RAM
4. Microphone

SOFTWARE:

1. Python and its libraries such as:
    - Librosa
    - Termcolor
    - NumPy
    - MatplotLib
    - SciPy
2. Ubuntu 18.04 LTS
3. DB Browser for SQLite

4. DESIGN APPROACH AND DETAILS

4.1. Design Approach and Methods:

4.1.1. Feature Extraction:

This project can be divided into various steps that take place in a specified order to achieve the final goal.

To start off, I decided to plot the amplitude of a song over time using matplotlib to get the following graph:

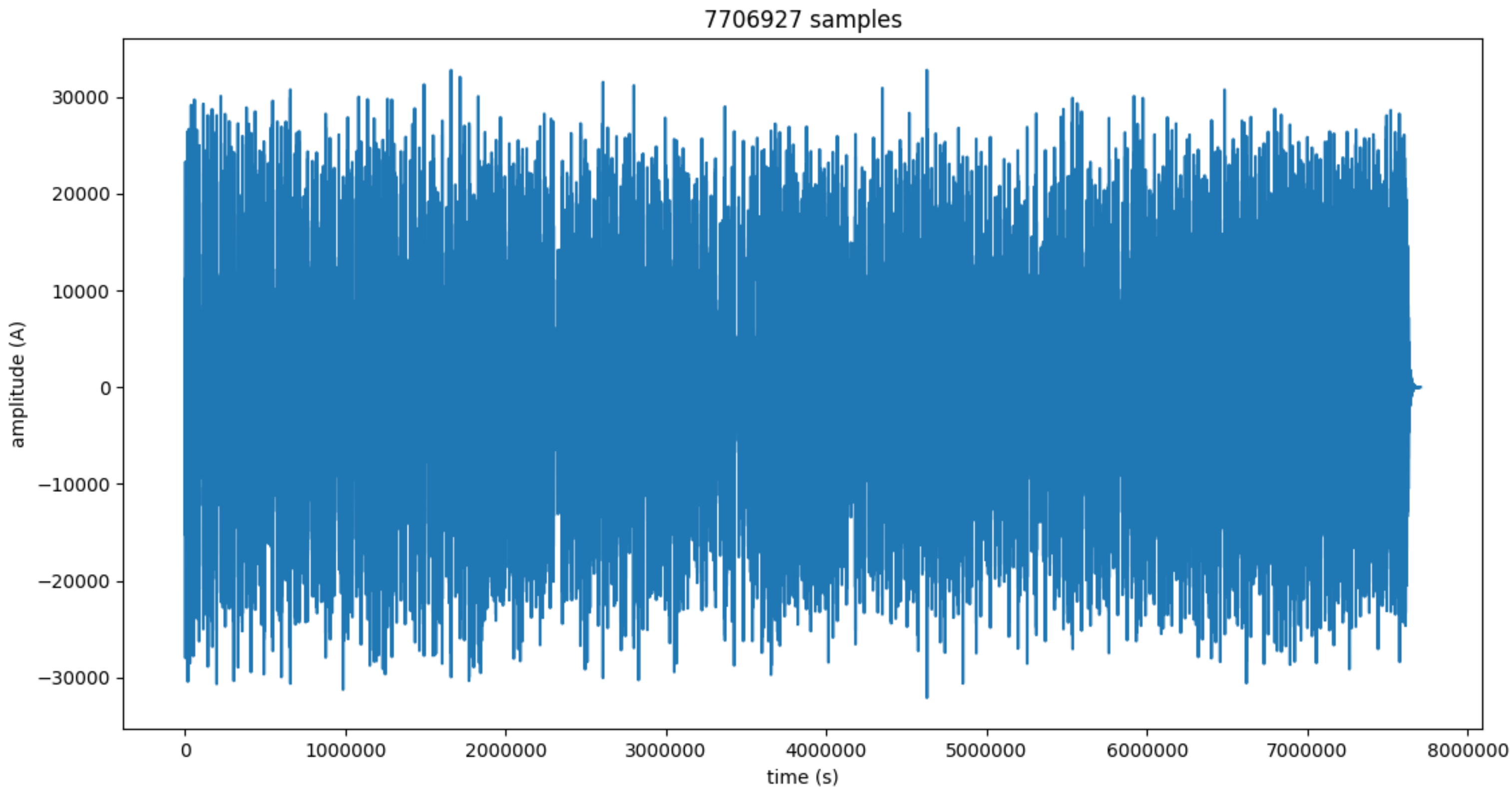


FIG 1: Amplitude vs Time

First, we need to decide on the feature that will be best for comparing with accuracy and speed being the main factors of deciding. I first decided to compare audio based on their feature known as Mel-frequency cepstrum (MFC).

MFC is a way of representing the pulse signals and energy signals at any given point of time of an audio signal. By calculating the MFCC at different points of time of the song (usually the minimum time between which a human ear can distinguish sounds), we get a long list of float values for every song.

It was a long and arduous process to convert every song to its MFCC values and store them in a database as a list. The retrieval would be very slow during matching and it would not be very efficient.

So, the next feature I decided to extract was harmonic and percussive nature of an audio file. While it is a very important feature for music classification and faster to calculate than MFCCs, they were not a good feature for matching audio. It was better for genre classification.

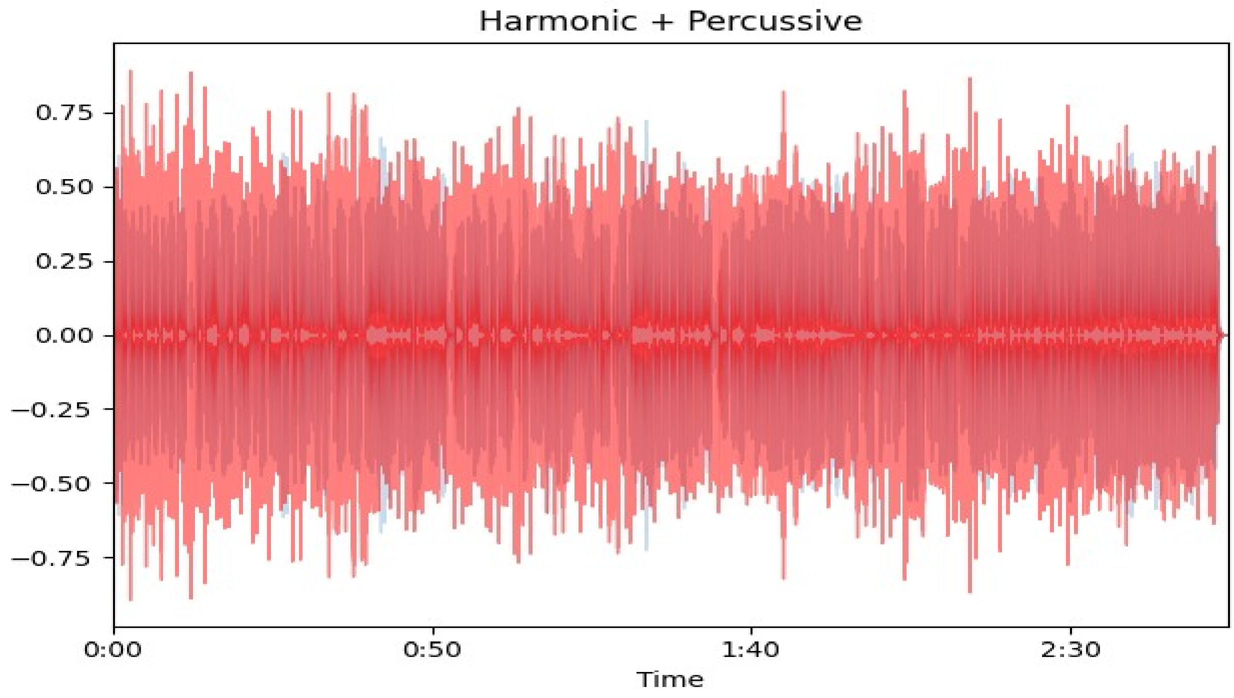


FIG 2: Harmonic and Percussive features

The final feature I decided to try, and the one used in this project, is the amplitude of the song wave over small time windows covering the entire song. On extracting the amplitude of every point and plotting the amplitude as a function of frequency and time. After plotting it we get the plot as a Spectrogram:

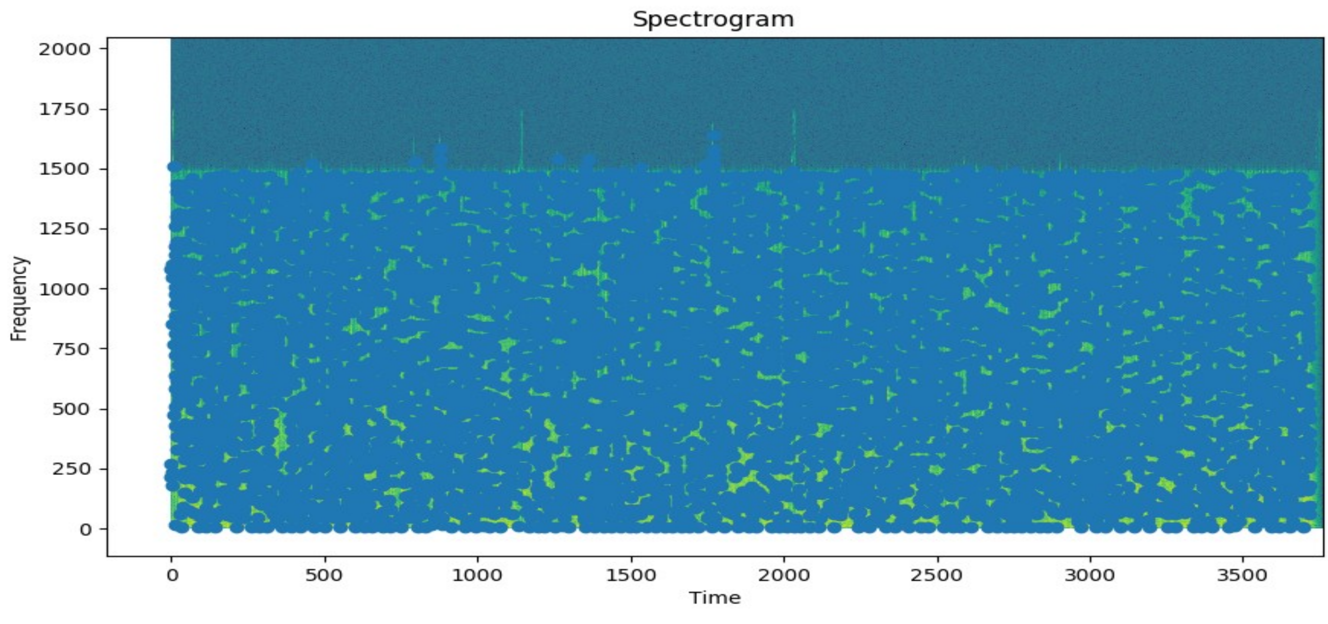


FIG 3: Spectrogram

Here then, I find out the peak of amplitude as a function of time and store that as the spectral centroid of that particular small window. The thought is that if amplitudes are matching then the audio signals, that is the song and music will match as well. This method is also much faster than calculating the MFCC of the small windows of the audio signal.

The spectrogram is create by applying FFT to small windows of time over the entire song.

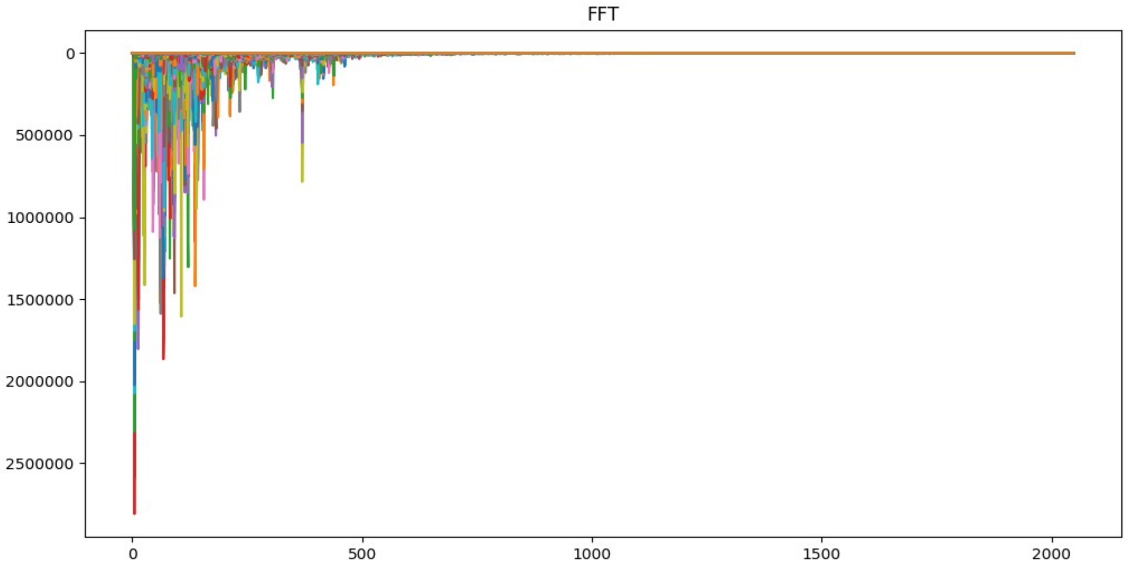


FIG 4: Fast Fourier Transform

The DFT (Discrete Fourier Transform) applies to discrete signals and gives a discrete spectrum (the frequencies inside the signal).

This is how we transform a digital signal to its frequencies:

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

In this formula:

- N is the size of the **window**: the number of samples that composed the signal (we'll talk a lot about windows in the next part).
- X(n) represents the nth **bin of frequencies**
- x(k) is kth sample of the audio signal

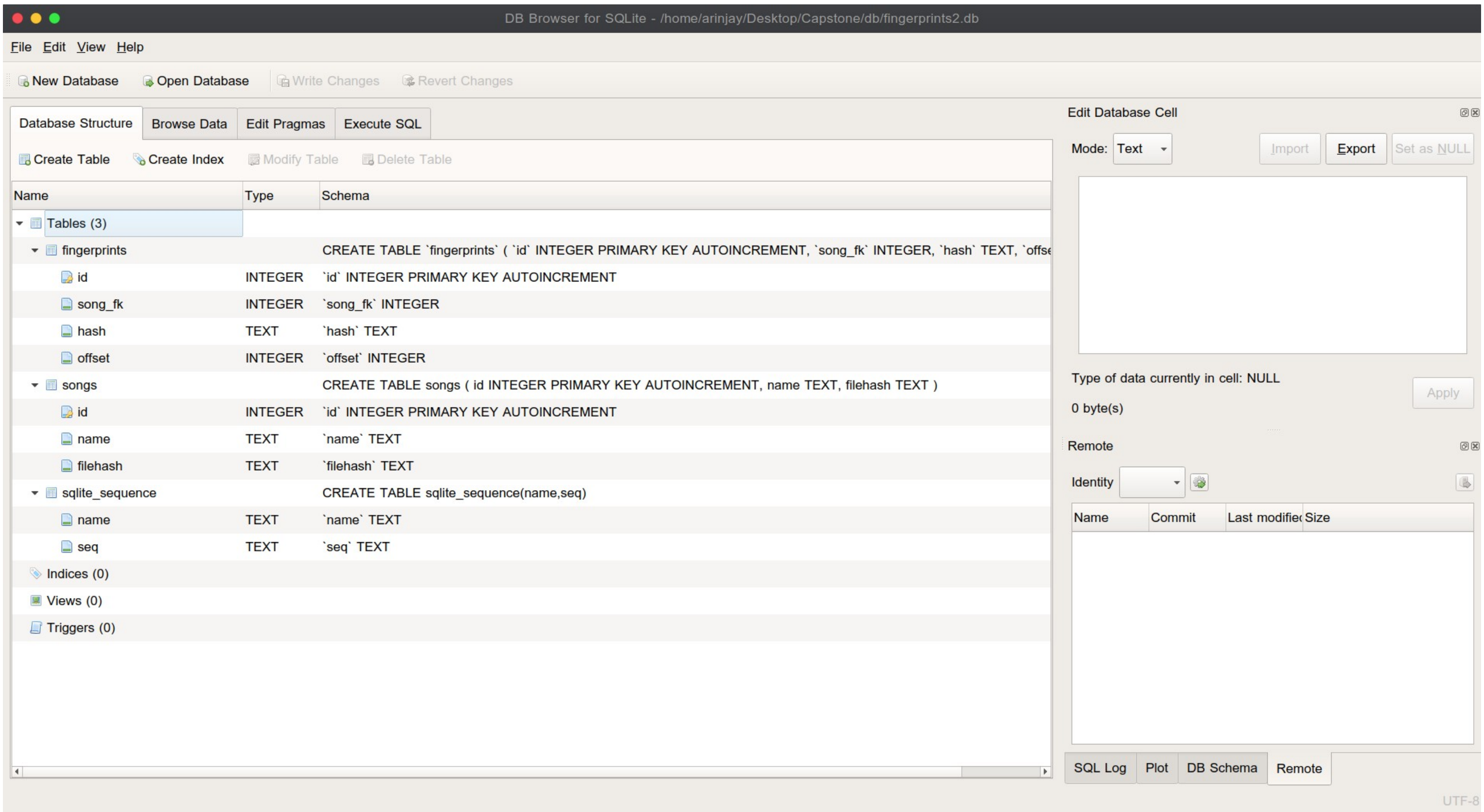
To make the storing of these values as well as the retrieval of these values during searching faster, I applied a hash function to them. Thus similar values will combine to form a “fingerprint”. These fingerprints are stored in the database.

Finally, during finding the match for a sample audio input, the input audio is also fingerprinted, and then its fingerprints are searched for in the database – which gives us the matching song from the database.

The output is decided by 2 factors – number of hash matches found and their offset.

4.1.2 Forming the Database

I used SQLite to create the database named "fingerprints.db". It has the following structure:



The Entity Relationship Diagram of the database is

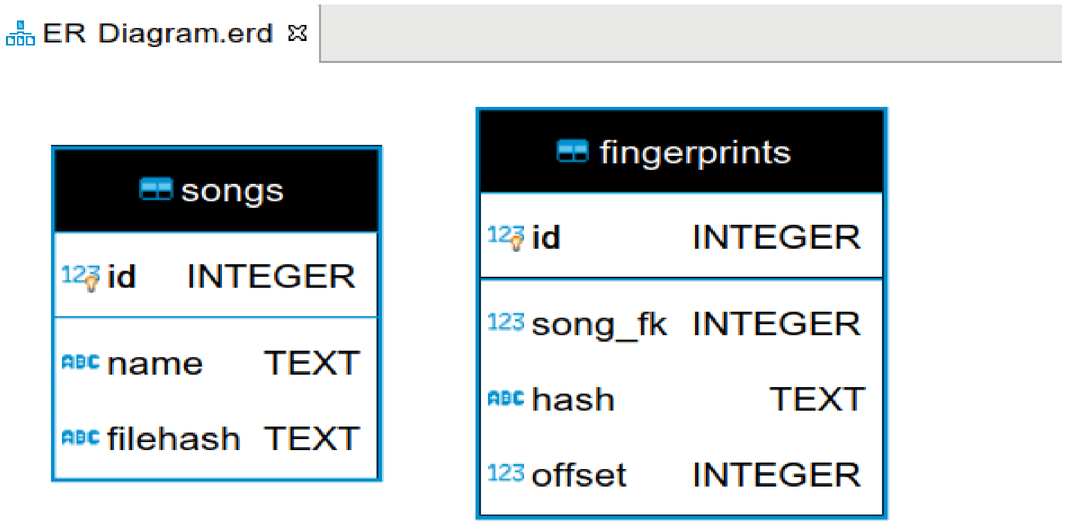
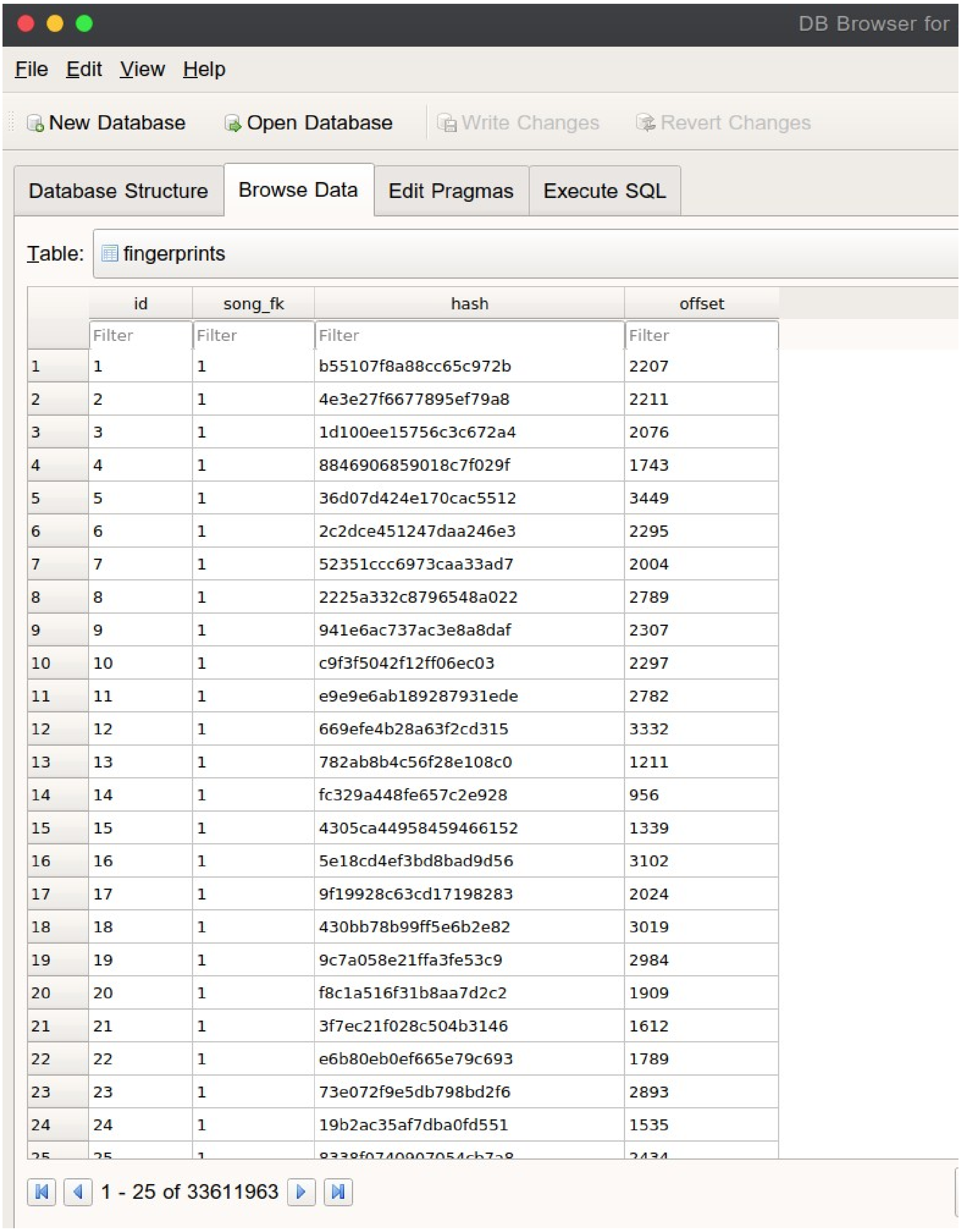


FIG 5: ER DIAGRAM

In the fingerprints table, we have a columns for song ID and hash values stored for that corresponding song. We also have an offset column which determines the time window of the corresponding hash value originated from. This is later useful for actually finding the correct hash values from all the matches we get on our input signal and the stored songs.

The table with values stored as



	id	song_fk	hash	offset
	Filter	Filter	Filter	Filter
1	1	1	b55107f8a88cc65c972b	2207
2	2	1	4e3e27f6677895ef79a8	2211
3	3	1	1d100ee15756c3c672a4	2076
4	4	1	8846906859018c7f029f	1743
5	5	1	36d07d424e170cac5512	3449
6	6	1	2c2dce451247daa246e3	2295
7	7	1	52351ccc6973caa33ad7	2004
8	8	1	2225a332c8796548a022	2789
9	9	1	941e6ac737ac3e8a8daf	2307
10	10	1	c9f3f5042f12ff06ec03	2297
11	11	1	e9e9e6ab189287931ede	2782
12	12	1	669efe4b28a63f2cd315	3332
13	13	1	782ab8b4c56f28e108c0	1211
14	14	1	fc329a448fe657c2e928	956
15	15	1	4305ca44958459466152	1339
16	16	1	5e18cd4ef3bd8bad9d56	3102
17	17	1	9f19928c63cd17198283	2024
18	18	1	430bb78b99ff5e6b2e82	3019
19	19	1	9c7a058e21ffa3fe53c9	2984
20	20	1	f8c1a516f31b8aa7d2c2	1909
21	21	1	3f7ec21f028c504b3146	1612
22	22	1	e6b80eb0ef665e79c693	1789
23	23	1	73e072f9e5db798bd2f6	2893
24	24	1	19b2ac35af7dba0fd551	1535

The songs table is pretty simple, we have a song ID automatically generated, the name of the song and the filehash of the particular song stored.

The table with song values stored is as follows

Database Structure | Browse Data | Edit Pragmas | Execute SQL

Table: songs

	id	name	filehash
	Filter	Filter	Filter
16	16	32. Santana with Rob Thomas - Smooth.mp3	7DA6C80437DCD078BECC1C680342342FEBD47942
17	17	44. Chris Rea - The Road To Hell Part 2.mp3	93ABFED0C64CB2D460654E1A0699163B77625A62
18	18	22. Snow Patrol - Chasing Cars.mp3	4E598A97FF138A3A41040A932B6F8BA141469662
19	19	15. INXS - Never Tear Us Apart.mp3	8356A2102302FC829B03EFC658C582D12796F2F7
20	20	02. The Police - Every Breath You Take.mp3	2CB8BAA7A283963A01D41AC16D7DA9E78FA8C716
21	21	38. Spin Doctors - Two Princes.mp3	3BF87DE792E6747854EEB1BAFAD7FBFC069E1F40
22	22	01. 4 Non Blondes - What's Up.mp3	AE0D94C609226A2EEC8DE1F06271E48A4D60CC81
23	23	48. Alanis Morissette - You Oughta Know (2015 Remaster).mp3	5B252FCD77020591EB4F7EC18B88F6C23C046F15
24	24	49. Styx - Babe.mp3	6E0F00945624DE1906BECE25FECC97E7E864F572
25	25	29. Peter Gabriel - Sledgehammer.mp3	32D31B4E207C205F57A87549F196DC38D6C4FF4B
26	26	50. Bryan Adams - (Everything I Do) I Do It For You.mp3	C450312622CBB16FEDECBB49FAD1430B2F23E79A
27	27	06. Journey - Don't Stop Believin'.mp3	0C0908D93B59BFC76E6529B3CA873AB31A14BB96
28	28	27. The Cars - Drive (2017 Remaster).mp3	2A2C6D0CA16CA13A7AE479E75831043FCFAC3ECA
29	29	36. Daniel Powter - Bad Day.mp3	67865E57A4744D1E90CAA6E5BB3508441308C79D
30	30	37. Genesis - Hold On My Heart.mp3	4BF7958CB32A797A13AEFCE3FE64C6B46EC5461C
31	31	21. Sixpence None The Richer - Kiss Me.mp3	E92886AE1326ABE373EC7434D575E6E3DC767FA8
32	32	05. R.E.M. - All The Way To Reno (You're Gonna Be A Star).mp3	F09EF1967A8C4A48462109DD7FD2C6BF4767A18B
33	33	33. James Blunt - 1973.mp3	5114BC7C8807A2A55EA63F13E99AD81E2B07C050
34	34	11. Travis - Sing.mp3	02AA8DF7119BB16A0229458017BFF460D2870E6B
35	35	41. Billy Joel - New York State Of Mind.mp3	F555DC02643F176371C1AED23C46D11CC1C01B36
36	36	40. KT Tunstall - Suddenly I See.mp3	D9C73CCA46877E6516A294024AC45830C930AD92
37	37	16. Kansas - Dust In The Wind.mp3	7697978AA688E69549BD7767874E11A5E82A862C
38	38	35. Fleetwood Mac - Everywhere (2017 Remaster).mp3	D92AA216A39CDCF24ACCDE29FB81402003A2DC60
39	39	13. Counting Crows - Mr. Jones.mp3	43D7E0EF0F812ADE1CF9D3212F194F659CC9E735
40	40	31. Eric Clapton - Wonderful Tonight.mp3	[illegible]

16 - 40 of 176 | Go to: 1

4.2. CODES AND STANDARDS

<u>To find the features and store them in a list in JSON format</u>

```
import sys
import json
import signal
import os.path
import argparse
import datetime
import librosa
import librosa.display
import numpy as np
import matplotlib.pyplot as plt

parser = argparse.ArgumentParser()

parser.add_argument('input', help='Directory with audio files to be analyzed (read is
recursive)')

parser.add_argument('-o', '--output', help='Optional path to json output (defaults to current
directory)')

args = parser.parse_args()

def signal_handler(signum, frame):
    print('Interrupted')
    sys.exit(0)
```

First, we import the Librosa, NumPy and Matplotlib libraries and take the input of the directory of the music files we want to extract features from.

The analyze function is responsible for loading the audio file and extracting their features and finally storing them into a list.

```
def analyze(files):
        output = []
   for file in files:

     y, sr = librosa.load(file)

      y_harm, y_perc = librosa.effects.hpss(y)

      plt.subplot(3, 1, 3)
      librosa.display.waveplot(y, sr=sr)
      plt.title('Stereo')
      plt.show()
```

First, for every audio file in the directory, it is first loaded as an audio waveform denoted by 'y', and the sampling rate is stored as 'sr'. I will discuss more about sampling rate later.

The **librosa.effects.hpss()** method divides the audio file into its harmonic and percussive components. I have then used Matplotlib to plot the graph of the amplitude of the audio file vs time (In stereo channel). This is stored as a .png file.

```
librosa.display.waveplot(y_harm, sr=sr, alpha=0.25)
librosa.display.waveplot(y_perc, sr=sr, color='r', alpha=0.5)
plt.title('Harmonic + Percussive')
plt.tight_layout()
plt.show()

X = librosa.stft(y)

Xdb = librosa.amplitude_to_db(abs(X))

plt.figure(figsize=(14, 5))
librosa.display.specshow(Xdb, sr=sr, x_axis='time', y_axis='hz')
plt.show()
```

The waveform is then created separately for the percussive and harmonic features of the audio file. It is then plotted on the graph and shown using Matplotlib.

The **librosa.stft()** method is responsible for applying Discrete Fourier Transform on very small overlapping windows of time. This allows us to get peaks of the amplitude which is then converted into a spectrogram using the **librosa.amplitude_to_db()** method. The spectrogram is then plot using Matplotlib.

```
spectral_centroids = librosa.feature.spectral_centroid(y=y, sr=sr)

spectral_bandwidths = librosa.feature.spectral_bandwidth(y=y, sr=sr)

S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128)

mfccs = librosa.feature.spectral.mfcc(y=y, sr=sr, S=librosa.amplitude_to_db(S),
n_mfcc=12)

duration = librosa.core.get_duration(y=y, sr=sr)
```

Using the functions in the Librosa library, different features such as spectral centroids, spectral bandwidths, MFCCs and duration of the song are extracted from the input audio music file and stored in a list.

```
json_data = {
        'path': file,
        'duration': duration,
        'spectral_centroids': spectral_centroids[0].tolist(),
        'spectral_bandwidths': spectral_bandwidths[0].tolist(),
        'spectral_centroid_min': spectral_centroids.min(),
        'spectral_centroid_max': spectral_centroids.max(),
        'spectral_centroid_mean': spectral_centroids.mean(),
        'mfccs': mfccs.tolist(),
    }

    print('Analyzing file:', file)
    output.append(json_data)
return output
```

The information that was stored in the list is then formatted into a dictionary which will be stored as a JSON format.

The following is the driver coded for reading the audio files from specified directory

```
if not args.input:
      print('No input path specified, see --help')
    sys.exit()

valid_extensions = ['aac', 'au', 'flac', 'm4a', 'mp3', 'ogg', 'wav', 'aif']
audio_file_path = os.path.expanduser(args.input)

if os.path.isdir(audio_file_path):
    audio_files = librosa.util.find_files(audio_file_path,  ext=valid_extensions)
else:
    audio_files = [audio_file_path]

result = analyze(audio_files)

def parse_input():

    yes = set(['yes', 'y', 'ye', ''])
    no = set(['no', 'n'])
    choice = input('-> ').lower()
    while True:
        if choice in yes:
            return True
        elif choice in no:
            return False
        else:
            sys.stdout.write("Please respond with 'yes' or 'no'")
```

The output is then written to the JSON file, if the file exists the user can decide whether to overwrite it or not.

```
def write_file(path, data):

    with open(path, 'w') as file:
        json.dump(data, file, separators=(',', ':'))
    print('Wrote output to', path)

if args.output:
    json_path = os.path.expanduser(args.output)

else:
    json_path = os.path.abspath('./output.json')

json_output = {
    'files': result,
    'timestamp': datetime.datetime.now().isoformat(),
    'version': '0.1'
}
if os.path.isfile(json_path):
    print('File {0} exists'.format(json_path))
    print('Overwrite?\n y/n')
    overwrite = parse_input()
    if overwrite:
        write_file(json_path, json_output)
    else:
        print('File was not overwritten')
else:
    write_file(json_path, json_output)
```

To Create Database and methods to add Audio Fingerprints to it

I have used the SQLite database management system as it is easy to implement and easy to access and retrieve data from using Python. The **sqlite3** library for Python allows me to achieve my goals.

```
from libs.db_sqlite import SqliteDatabase

if __name__ == '__main__':
    db = SqliteDatabase()

    db.query("DROP TABLE IF EXISTS songs;")
    print('removed db.songs')

    db.query("""
    CREATE TABLE songs (
      id  INTEGER PRIMARY KEY AUTOINCREMENT,
      name  TEXT,
      filehash  TEXT
    );
  """)
    print('created db.songs')

    db.query("DROP TABLE IF EXISTS fingerprints;")
    print('removed db.fingerprints')

    db.query("""
    CREATE TABLE `fingerprints` (
      `id`  INTEGER PRIMARY KEY AUTOINCREMENT,
      `song_fk` INTEGER,
      `hash`  TEXT,
      `offset`  INTEGER
    );
  """)
    print('created db.fingerprints')

    print('done')
```

For that I run the reset-database.py file which will delete the table if it already exists and then create new instances of the tables with no values that we can insert into.

```python
from db import Database
from config import get_config
import sqlite3
import sys
from itertools import izip_longest
from termcolor import colored


class SqliteDatabase(Database):
    TABLE_SONGS = 'songs'
    TABLE_FINGERPRINTS = 'fingerprints'

    def __init__(self):
        self.connect()

    def connect(self):
        config = get_config()

        self.conn = sqlite3.connect(config['db.file'])
        self.conn.text_factory = str

        self.cur = self.conn.cursor()

        print(colored('sqlite - connection opened', 'white', attrs=['dark']))

    def __del__(self):
        self.conn.commit()
        self.conn.close()
        print(colored('sqlite - connection has been closed', 'white', attrs=['dark']))
```

Here, I have created an instance of my SQLite database and its Tables. On being called, the **db.connect()** method is called which establishes the connection between the program and the database. After the query has been executed – for adding songs to database, removing songs from the database, or just finding a match between fingerprints – the connection is closed with the use of **db.conn.close()** method

Next, for query there have to be methods that can take a string as argument and pass that as a query to the database. The executeOne and executeAll methods will execute the query for one row or all rows in the database.

```
def query(self, query, values=[]):
    self.cur.execute(query, values)

def executeOne(self, query, values=[]):
    self.cur.execute(query, values)
    return self.cur.fetchone()

def executeAll(self, query, values=[]):
    self.cur.execute(query, values)
    return self.cur.fetchall()

def buildSelectQuery(self, table, params):
    conditions = []
    values = []

    for k, v in enumerate(params):
        key = v
        value = params[v]
        conditions.append("%s = ?" % key)
        values.append(value)

    conditions = ' AND '.join(conditions)
    query = "SELECT * FROM %s WHERE %s" % (table, conditions)

    return {
        "query": query,
        "values": values
    }
```

The findOne and findAll methods also have the same task, they just return a single value or multiple values according to the method. The insert method takes a table and query as arguments and then inserts the song into the songs table of the database. It then returns the row where the song was inserted.

```python
def findOne(self, table, params):
    select = self.buildSelectQuery(table, params)
    return self.executeOne(select['query'], select['values'])

def findAll(self, table, params):
    select = self.buildSelectQuery(table, params)
    return self.executeAll(select['query'], select['values'])

def insert(self, table, params):
    keys = ', '.join(params.keys())
    values = params.values()

    query = "INSERT INTO songs (%s) VALUES (?, ?)" % (keys);

    self.cur.execute(query, values)
    self.conn.commit()

    return self.cur.lastrowid
```

The insertMany method allows us to insert multiple songs at once into the songs database from a single directory. The get_song_hashes_count method calculates the number of hash values generated for a single song from the fingerprints table. The song ID is compared in both the tables and number of hash values is then returned.

```python
def insertMany(self, table, columns, values):
    def grouper(iterable, n, fillvalue=None):
        args = [iter(iterable)] * n
        return (filter(None, values) for values
                in izip_longest(fillvalue=fillvalue, *args))

    for split_values in grouper(values, 1000):
        query = "INSERT OR IGNORE INTO %s (%s) VALUES (?, ?, ?)" % (table, ", ".join(columns))
        self.cur.executemany(query, split_values)

    self.conn.commit()

def get_song_hashes_count(self, song_id):
    query = 'SELECT count(*) FROM %s WHERE song_fk = %d' % (self.TABLE_FINGERPRINTS, song_id)
    rows = self.executeOne(query)
    return int(rows[0])
```

Creating the Fingerprint for the audio file

```
import hashlib
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
from termcolor import colored
from scipy.ndimage.filters import maximum_filter
from scipy.ndimage.morphology import (generate_binary_structure, iterate_structure, binary_erosion)
from operator import itemgetter

IDX_FREQ_I = 0
IDX_TIME_J = 1

DEFAULT_FS = 44100

DEFAULT_WINDOW_SIZE = 4096

DEFAULT_OVERLAP_RATIO = 0.5

DEFAULT_FAN_VALUE = 15

DEFAULT_AMP_MIN = 10

PEAK_NEIGHBORHOOD_SIZE = 20

MIN_HASH_TIME_DELTA = 0

MAX_HASH_TIME_DELTA = 200

PEAK_SORT = True

FINGERPRINT_REDUCTION = 20
```

The DEFAULT_FS is the sampling rate of the audio input, it is determined by the Nyquist conditions and chosen to detect the range of frequencies we can detect.

In the case of recording audio, we do not consider frequencies above 22050 Hz since humans can't hear frequencies above 20,000 Hz. Thus by Nyquist, we have to sample *twice* that:

Samples per sec needed = Highest-Frequency * 2 = 22050 * 2 = 44100

The DEFAULT_WINDOW_SIZE for applying FFT is chosen as 4096 in order to reduce frequency granularity.

The DEFAULT_OVERLAP_RATIO is the ratio by which each sequential window overlaps the last and the next window. Higher overlap will allow a higher granularity of offset matching, but potentially more fingerprints.

The DEFAULT_FAN_VALUE is the degree to which a fingerprint can be paired with a nearby neighbor. A higher value will result in more fingerprints being formed but will increase accuracy as well.

The DEFAULT_AMP_MIN is the minimum amplitude in the spectrogram that will be considered to be a peak.

The PEAK_NEIGHBOURHOOD_SIZE is the number of cells around an amplitude for it to be considered a spectral peak.
The MIN and MAX HASH_TIME_DELTA are thresholds on how close or far fingerprints can be in time in order to be paired as a fingerprint.
The FINGERPRINT_REDUCTION is the number of bits to throw away from the front of the SHA1 hash in the fingerprint calculation. Higher value means lower storage requirements in database but will result in more chances of misclassification.

```
def fingerprint(channel_samples, Fs=DEFAULT_FS, wsize=DEFAULT_WINDOW_SIZE,
                wratio=DEFAULT_OVERLAP_RATIO, fan_value=DEFAULT_FAN_VALUE,
                amp_min=DEFAULT_AMP_MIN):
    arr2D = mlab.specgram(
        channel_samples,
        NFFT=wsize,
        Fs=Fs,
        window=mlab.window_hanning,
        noverlap=int(wsize * wratio))[0]

    arr2D = 10 * np.log10(arr2D)
    arr2D[arr2D == -np.inf] = 0

    local_maxima = get_2D_peaks(arr2D, plot=plots, amp_min=amp_min)

    msg = '   local_maxima: %d of frequency & time pairs'
    print colored(msg, attrs=['dark']) % len(local_maxima)

    return generate_hashes(local_maxima, fan_value=fan_value)
```

In the fingerprint method, we call the **mlab.specgram()** method from Matplotlib to create our Spectrogram for the audio file we are processing.

As specgram returns a linear array, we apply log transform on it. We calculate the base 10 log values of all the elements in the array and replace infinity with 0.

The next step is to find the local maxima in the window and then return the hashes.

```
def get_2D_peaks(arr2D, plot=False, amp_min=DEFAULT_AMP_MIN):
    struct = generate_binary_structure(2, 1)
    neighborhood = iterate_structure(struct, PEAK_NEIGHBORHOOD_SIZE)

    local_max = maximum_filter(arr2D, footprint=neighborhood) == arr2D
    background = (arr2D == 0)
    eroded_background = binary_erosion(background, structure=neighborhood,
                                       border_value=1)

    detected_peaks = local_max ^ eroded_background

    amps = arr2D[detected_peaks]
    j, i = np.where(detected_peaks)

    amps = amps.flatten()
    peaks = zip(i, j, amps)
    peaks_filtered = [x for x in peaks if x[2] > amp_min]  # freq, time, amp

    frequency_idx = [x[1] for x in peaks_filtered]
    time_idx = [x[0] for x in peaks_filtered]
```

We use the **generate_binary_structure()** of SciPy to and iterate through it using the **iterate_structure()** method. Then we find the local maxima using the filter shape created above. We get a boolean mask with the “True” value at the peaks. After identifying the peaks, we extract the detected peaks and get their corresponding frequency and time indices and return the values.

```python
def generate_hashes(peaks, fan_value=DEFAULT_FAN_VALUE):
    if PEAK_SORT:
        peaks.sort(key=itemgetter(1))

    for i in range(len(peaks)):
        for j in range(1, fan_value):
            if (i + j) < len(peaks):

                freq1 = peaks[i][IDX_FREQ_I]
                freq2 = peaks[i + j][IDX_FREQ_I]

                t1 = peaks[i][IDX_TIME_J]
                t2 = peaks[i + j][IDX_TIME_J]

                t_delta = t2 - t1

                if t_delta >= MIN_HASH_TIME_DELTA and t_delta <= MAX_HASH_TIME_DELTA:
                    h = hashlib.sha1("%s|%s|%s" % (str(freq1), str(freq2), str(t_delta)))
                    yield (h.hexdigest()[0:FINGERPRINT_REDUCTION], t1)
```

The hash list structure is **sha1_hash[0:20], time_offset** for example: [(e05b341a9b77a51fd26, 32), ….]. Here we have no option but to bruteforce all the peaks. We take the current and next peak value, get their offset values and then calculate the “delta” value, i.e. the difference of time offsets of both peaks.

Next, we need to check if the difference is between the minimum and maximum HASH_TIME_DELTA. Yield the hashed value of the peaks. This value is stored in the fingerprint database.

To collect the fingerprints of a directory of songs and store in database

```
import os
import libs.fingerprint as fingerprint
from termcolor import colored
from libs.reader_file import FileReader
from libs.db_sqlite import SqliteDatabase
from libs.config import get_config

if __name__ == '__main__':
  config = get_config()

  db = SqliteDatabase()
  path = "mp3/"

  for filename in os.listdir(path):
    if filename.endswith(".mp3"):
      reader = FileReader(path + filename)
      audio = reader.parse_audio()

      song = db.get_song_by_filehash(audio['file_hash'])
      song_id = db.add_song(filename, audio['file_hash'])

      msg = ' * %s %s: %s' % (
        colored('id=%s', 'white', attrs=['dark']),       # id
        colored('channels=%d', 'white', attrs=['dark']), # channels
        colored('%s', 'white', attrs=['bold'])           # filename
      )
      print msg % (song_id, len(audio['channels']), filename)

      if song:
        hash_count = db.get_song_hashes_count(song_id)
```

Here, we first initialize the database and start reading all the audio files saved as .mp3 in the /mp3 directory. We hash its values on both stereo and mono channels and store it in the database.

```python
    if hash_count > 0:
      msg = '   already exists (%d hashes), skip' % hash_count
      print colored(msg, 'red')

      continue

  print colored('   new song, going to analyze..', 'green')

  hashes = set()
  channel_amount = len(audio['channels'])

  for channeln, channel in enumerate(audio['channels']):
    msg = '   fingerprinting channel %d/%d'
    print colored(msg, attrs=['dark']) % (channeln+1, channel_amount)

    channel_hashes = fingerprint.fingerprint(channel, Fs=audio['Fs'], plots=config['fingerprint.show_plots'])
    channel_hashes = set(channel_hashes)

    msg = '   finished channel %d/%d, got %d hashes'
    print colored(msg, attrs=['dark']) % (
      channeln+1, channel_amount, len(channel_hashes)
    )

    hashes |= channel_hashes

  msg = '   finished fingerprinting, got %d unique hashes'
```

The values of the hash and the offset are stored in a list. Then the values are appended and stored in the fingerprints table of the database.

```python
    values = []
    for hash, offset in hashes:
      values.append((song_id, hash, offset))

    msg = '   storing %d hashes in db' % len(values)
    print colored(msg, 'green')

    db.store_fingerprints(values)

print('end')
```

To listen to audio through microphone and find match in database

```
import sys
from matplotlib import pyplot
import libs.fingerprint as fingerprint
import argparse
from argparse import RawTextHelpFormatter
from itertools import izip_longest
from termcolor import colored
from libs.config import get_config
from libs.reader_microphone import MicrophoneReader
from libs.visualiser_console import VisualiserConsole as visual_peak
from libs.db_sqlite import SqliteDatabase

if __name__ == '__main__':
  config = get_config()

  db = SqliteDatabase()

  parser = argparse.ArgumentParser(formatter_class=RawTextHelpFormatter)
  parser.add_argument('-s', '--seconds', nargs='?')
  args = parser.parse_args()

  if not args.seconds:
    parser.print_help()
    sys.exit(0)

  seconds = int(args.seconds)

  chunksize = 2**12
  channels = 2
```

Here we define the window size as “chunksize” (2**12 = 4096), and channels as 2 so we can do both mono as well as stereo comparisons. We take an argument while executing the file of the number of seconds that the microphone should listen. On testing I found that a minimum of 5 seconds was needed to avoid false positives. The longer the recording the higher the chances of getting a correct match.

We also initialize an instance of our SQLite database here with which we will match the recorded audio fingerprints to find a match.

```
record_forever = False
visualise_console = bool(config['mic.visualise_console'])
visualise_plot = bool(config['mic.visualise_plot'])

reader = MicrophoneReader(None)

reader.start_recording(seconds=seconds,
  chunksize=chunksize,
  channels=channels)

msg = ' * started recording..'
print colored(msg, attrs=['dark'])

while True:
  bufferSize = int(reader.rate / reader.chunksize * seconds)

  for i in range(0, bufferSize):
    nums = reader.process_recording()

    if visualise_console:
      msg = colored('   %05d', attrs=['dark']) + colored(' %s', 'green')
      print msg  % visual_peak.calc(nums)
    else:
      msg = '   processing %d of %d..' % (i, bufferSize)
      print colored(msg, attrs=['dark'])

  if not record_forever: break
```

Here we start recording the audio for the specified time (in seconds) and have a small visualization of the amplitude of the input audio display as a plot in the terminal itself. The buffer size is determined with the rate of input stream, the chunk size initialized and the number of seconds that were passed as argument.

```
    msg = ' * recorded %d samples'
    print colored(msg, attrs=['dark']) % len(data[0])

    Fs = fingerprint.DEFAULT_FS
    channel_amount = len(data)

    result = set()
    matches = []


def find_matches(samples, Fs=fingerprint.DEFAULT_FS):
    hashes = fingerprint.fingerprint(samples, Fs=Fs)
    return return_matches(hashes)
```

After the audio is recorded, a result set and a matches list is initialized. This is where we will store all the matches we get from the database for the input audio stream.

The find_matches() method will return all the matches that were found by the return_matches() method that is below.

Here we make use of the offset value. This offset in timing can be calculated by subtracting the time of the anchor-point pair's occurrence in the input audio's recording from the matching hash's time of occurrence in the audio file from the stored database. If a significant amount of matching hashes have the same time offset, that song is determined to be a match.

```python
def return_matches(hashes):
    mapper = {}
    for hash, offset in hashes:
        mapper[hash.upper()] = offset
    values = mapper.keys()

    for split_values in grouper(values, 1000):
        # @todo move to db related files
        query = """
    SELECT upper(hash), song_fk, offset
    FROM fingerprints
    WHERE upper(hash) IN (%s)
  """
        query = query % ', '.join('?' * len(split_values))

        x = db.executeAll(query, split_values)
        matches_found = len(x)

        if matches_found > 0:
            msg = '    ** found %d hash matches (step %d/%d)'
            print colored(msg, 'green') % (
                matches_found,
                len(split_values),
                len(values)
            )
        else:
            msg = '    ** not matches found (step %d/%d)'
            print colored(msg, 'red') % (
                len(split_values),
                len(values)
            )
```

After we find the matching hash and offset values, we execute a query to select the song ID of the correct match from the fingerprints table. In case no match is found, we give a negative output.

```python
def align_matches(matches):
    diff_counter = {}
    largest = 0
    largest_count = 0
    song_id = -1

    for tup in matches:
        sid, diff = tup

        if diff not in diff_counter:
            diff_counter[diff] = {}

        if sid not in diff_counter[diff]:
            diff_counter[diff][sid] = 0

        diff_counter[diff][sid] += 1

        if diff_counter[diff][sid] > largest_count:
            largest = diff
            largest_count = diff_counter[diff][sid]
            song_id = sid

    songM = db.get_song_by_id(song_id)

    nseconds = round(float(largest) / fingerprint.DEFAULT_FS *
                     fingerprint.DEFAULT_WINDOW_SIZE *
                     fingerprint.DEFAULT_OVERLAP_RATIO, 5)
```

We find out the correct match by using the time offset values and the differences of offset values to find the correct song from the database. Once we have our match, we retrieve the song ID, song name, offset and the confidence of the match and give them as output.

```python
    return {
        "SONG_ID": song_id,
        "SONG_NAME": songM[1],
        "CONFIDENCE": largest_count,
        "OFFSET": int(largest),
        "OFFSET_SECS": nseconds
    }


total_matches_found = len(matches)

print ''

if total_matches_found > 0:
    msg = ' ** totally found %d hash matches'
    print colored(msg, 'green') % total_matches_found

    song = align_matches(matches)

    msg = ' => song: %s (id=%d)\n'
    msg += '    offset: %d (%d secs)\n'
    msg += '    confidence: %d'

    print colored(msg, 'green') % (
        song['SONG_NAME'], song['SONG_ID'],
        song['OFFSET'], song['OFFSET_SECS'],
        song['CONFIDENCE']
    )
else:
    msg = ' ** not matches found at all'
    print colored(msg, 'red')
```

4.3. CONSTRAINTS, ALTERNATIVES AND TRADEOFFS

Constraints

1. As we are using audio fingerprinting, it is not very robust in finding a correct matching output if the input audio is not very close to the stored fingerprints in the database.
For example, if a person played his own cover for a song such as “Bohemian Rhapsody”, the difference in amplitudes may result in a different spectrogram and ultimately in a false positive output, i.e it will match with some other audio that is “more similar” to it in the sequence of amplitudes than the original Bohemian Rhapsody song as recorded by QUEEN.

2. Another constraint was the time of matching. For a very large database, a more efficient way of storing and retrieving fingerprints must be implemented.

Alternatives

There are different ways to match audio files from a database and an input audio file that do not follow the fingerprinting method. There might even be ways to match audio songs using “humming” where the input is a person humming the tune of the song directly from their mouth and using that as input.

Tradeoffs

One of the biggest tradeoffs to reduce the size of the database was using the audio files as .mp3 instead of the full .wav files. The difference is that .mp3 files are compressed audio files and have a much smaller size than the lossless .wav files – which have a much higher quality. Thus for the database .mp3 format audio files were used to keep the storage size lower than it could have been. For about 40 files I downloaded the .mp3 and .wav files. The difference in storage was as follows:

TABLE 2: Storage of Music and FIngerprints

File Type	Storage (in MB)
.mp3	339
.wav	1885
Fingerprint	337

We can see that the size difference between the .mp3 files and .wav files is almost 6 times.

5. SCHEDULE, TASKS AND MILESTONES

The schedule I followed for my capstone project was basically doing tasks to reach various milestones to complete the program and get the desired output.

1. Deciding Topic: The first step was actually decide on a topic that had practical application and would allow me to improve my own skill set.

2. Research: The most important step was researching, I had to go through lot of different websites to learn about how Python deals with audio, and then I had to peruse various academic papers related to music and sound analysis. I learned the different approaches that people have used in order to match music based on feature extraction and using Machine Learning techniques.

3. Basic Implementation: I then decided to find a simpler way of implementing a music similarity checker. Python libraries such as Librosa, SciPy, numpy and more helped me decide on my own method of approach.

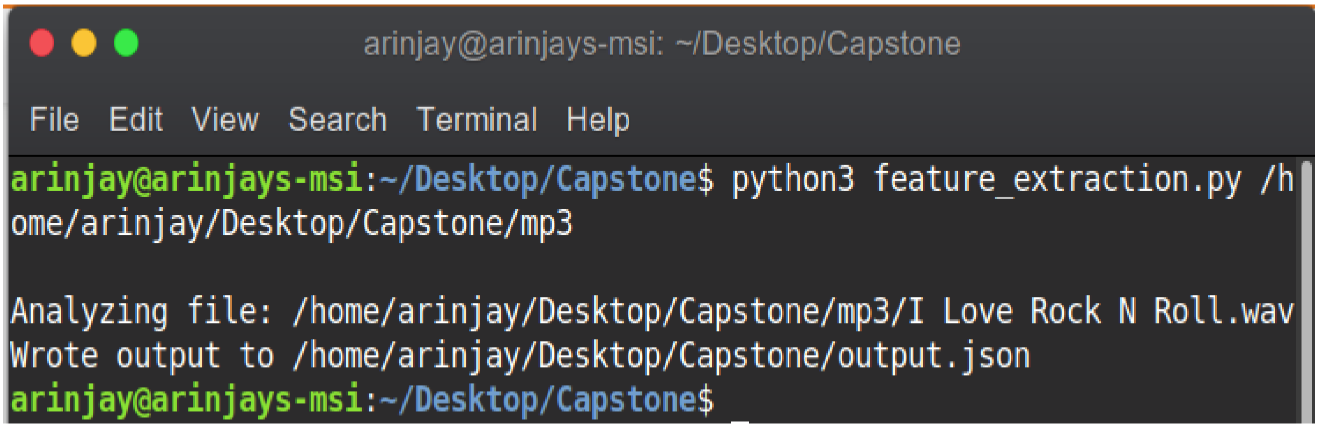
4. Implementation: The end of this step was my first milestone. This is when I started implementing the code that was finally executed. I had to read through the library docs and finally implemented the project in Linux distribution Ubuntu 18.04 instead of Windows 10 as it was easier to manage python environments and installation of libraries.

5. Creation of the Database: The final step was to create the database and store the songs and their fingerprints in the tables I had created. For this step I downloaded a lot of .mp3 files and fingerprinted them using my project.

6. Testing: The final step was testing the accuracy and efficiency of my project and to note them down as results.

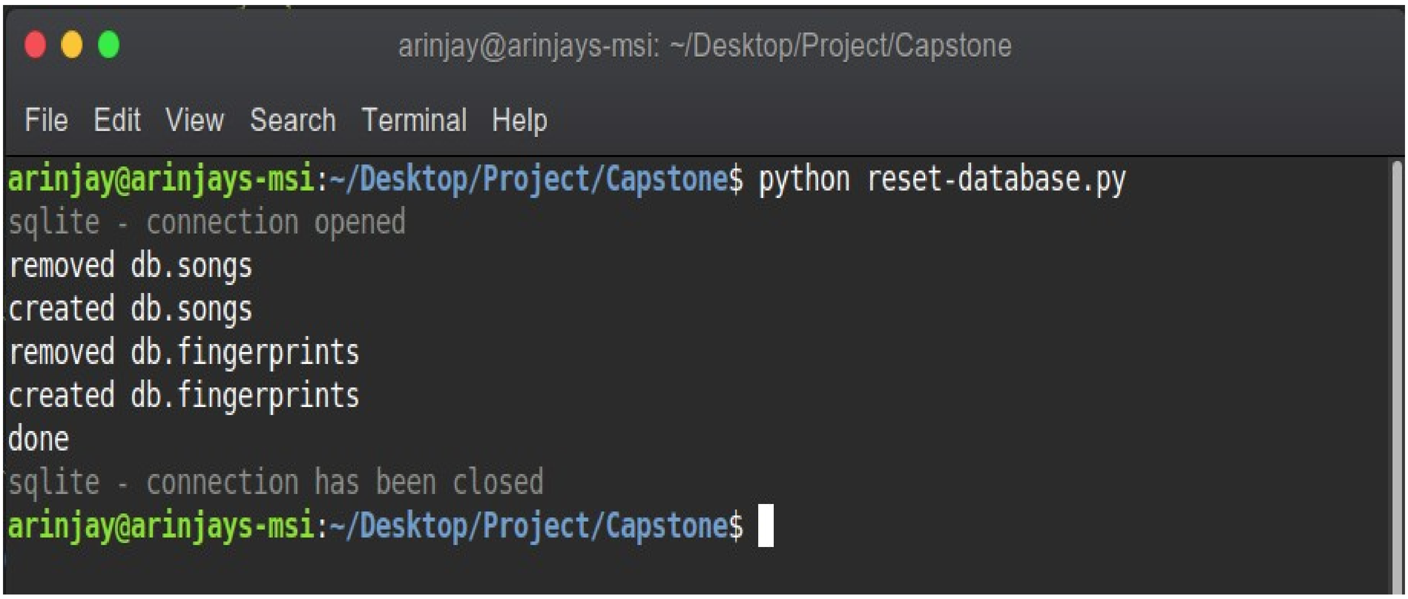
6. PROJECT DEMONSTRATION AND RESULT

When I was testing Librosa module to test the different features that can be extracted from the audio file and stored as a list in JSON format, the following command was run



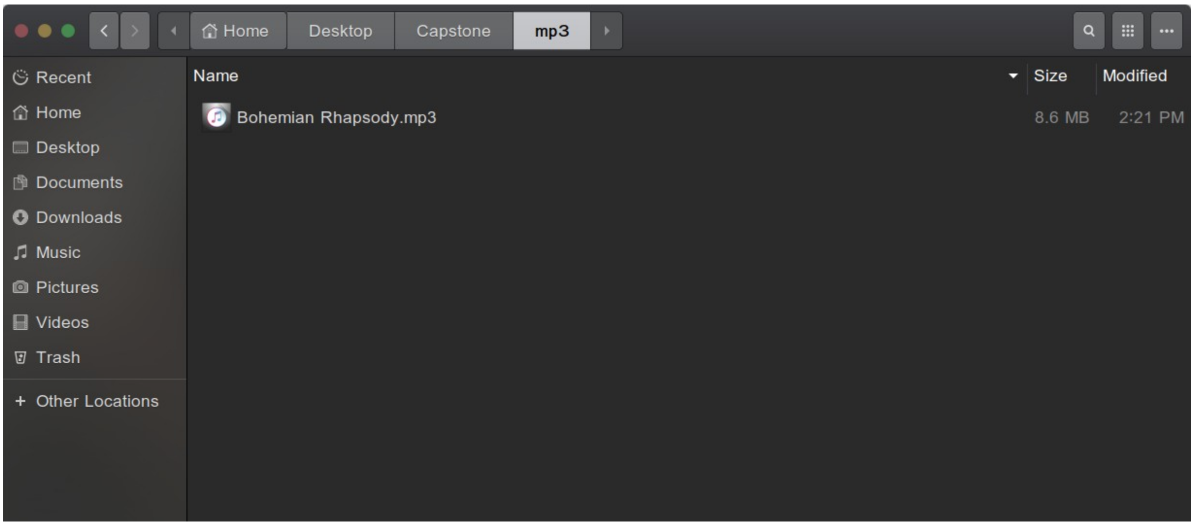
This gave the output as a JSON file along with the graphs for stereo, harmonic and percussive as well as a spectrogram.

For the final implementation, first we need to create our tables in SQLite.
On running the python file reset-database.py , we get the following output

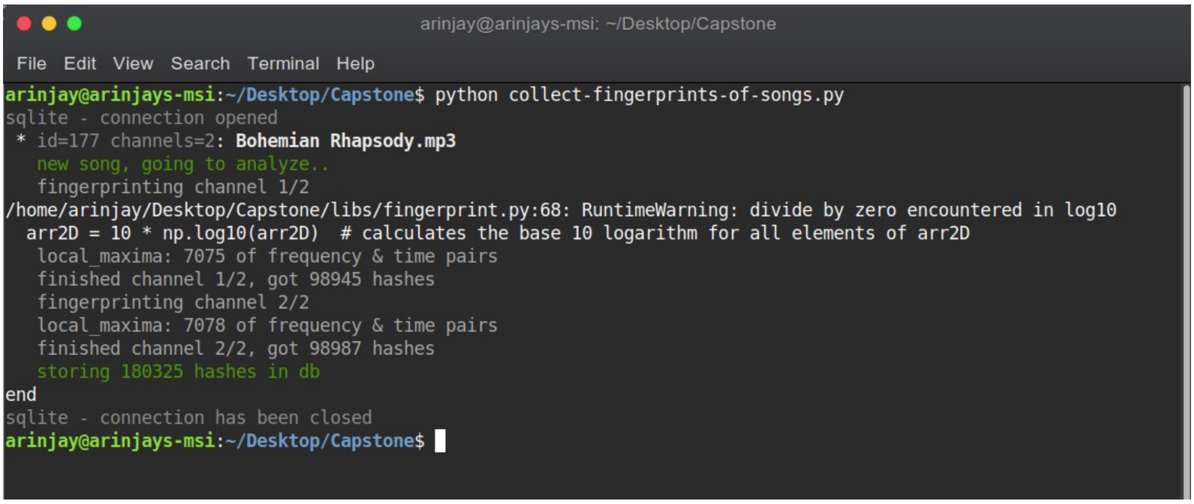


And we can see the database being created on the DB Browser for SQLite application available for Ubuntu.

The next step is to store some .mp3 files in the '/mp3' directory in the root project folder. We will fingerprint all of those songs and store their hash values and fingerprints in the database for later.



For the demonstration purposes, I have only included a single song in the /mp3 directory. It is 'Bohemian Rhapsody' by QUEEN. The song has a very wide variety of sounds in it. When we run the



Now we can check the DB Browser for SQLite and check for ourselves if the song "Bohemian Rhapsody" is in the songs table with id = 177.

176	176	I Love Rock N Roll.mp3	464E21578678EBFF7AFCC1C2C533D94675...
177	177	Bohemian Rhapsody.mp3	4A4278909D26955BAD1EB4D9AD37FE316...

Here we can see the song id 177 has name Bohemian Rhapsody and that its filehash value is 4A4278909D26955BAD1EB4D9AD37FE316C7E32BB.

We can even open the fingerprints table and see the values stored corresponding to song id 177 with the use of DB Browser for SQLite.

Table: fingerprints | New Record | De

	id	song_fk	hash	offset
	Filter	Filter	Filter	Filter
2	33792287	177	e456afaf581e84cf2f85	6116
3	33792286	177	0c7183db7490a1f6b1aa	3276
4	33792285	177	00e7334dc8319549f322	6415
5	33792284	177	07595e9a9ec3976aa052	4943
6	33792283	177	c47dfd90a3d47883c4d5	3176
7	33792282	177	5c8b2090e169b0a195d6	2892
8	33792281	177	b36a5cc9c2f95fbf4eb3	5309
9	33792280	177	da3cad69d959e8a8932f	7484
10	33792279	177	0240b04b68eb1aaddb23	3005
11	33792278	177	b0062a6bf2720ece3ab8	4239
12	33792277	177	22130771e5b91b6111b7	946
13	33792276	177	7b82c58e18fc563d92fa	5797
14	33792275	177	1a9ebc8f6500e63fa972	6736
15	33792274	177	e7e52e1ce342fdfcb43c	1814
16	33792273	177	d338c743525e5c5e6ff0	2403
17	33792272	177	254d2af36037fdba7005	4830
18	33792271	177	66cb76a55828dc102751	435
19	33792270	177	fe3a0a550fc515fa1ffa	3354
20	33792269	177	78772dc8e03792e29ac4	5678
21	33792268	177	c29fbbe5c63b3746761d	3687
22	33792267	177	e538bac22cc3baf03aa5	2223
23	33792266	177	0877d3b8196e68fc647b	1680
24	33792265	177	183fe13e8c2e3eb85ba1	6470
25	33792264	177	35d5876b29a17eead796	1210
26	33792263	177	[illegible]	[illegible]

2 - 26 of 33792288 | Go to: 1

We can clearly see here the different hash values stored for the song id 177 along with their offset values in the fingerprints table.

The next task to perform is recording an input of the song and getting an output for it from the database. For that we have to run the recognize-from-microphone.py file along with a parameter of the number of seconds to record. For the purpose of the demonstration I have recorded 10 seconds of a random part from the song “Bohemian Rhapsody” from Spotify.

The green bars give an approximate visual representation of the amplitude or loudness of the audio file at that moment of time.

```
arinjay@arinjays-msi: ~/Desktop/Capstone
File Edit View Search Terminal Help
arinjay@arinjays-msi:~/Desktop/Capstone$ python recognize-from-microphone.py -s 10
sqlite - connection opened
ALSA lib pcm_dmix.c:1052:(snd_pcm_dmix_open) unable to open slave
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM cards.pcm.rear
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM cards.pcm.center_lfe
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM cards.pcm.side
ALSA lib pcm_route.c:867:(find_matching_chmap) Found no matching channel map
ALSA lib pcm_dmix.c:1052:(snd_pcm_dmix_open) unable to open slave
 * started recording..
   18631 ########################################################
   11378 ##################################
   07044 #####################
   04358 #############
   03259 #########
   03663 ###########
   02979 #########
   04004 ############
   03848 ###########
   03154 #########
   01813 #####
   00914 ##
   00779 ##
   00793 ##
   01312 ####
   01073 ###
   01122 ###
   01348 ####
   01573 ####
   01248 ###
   01189 ###
```

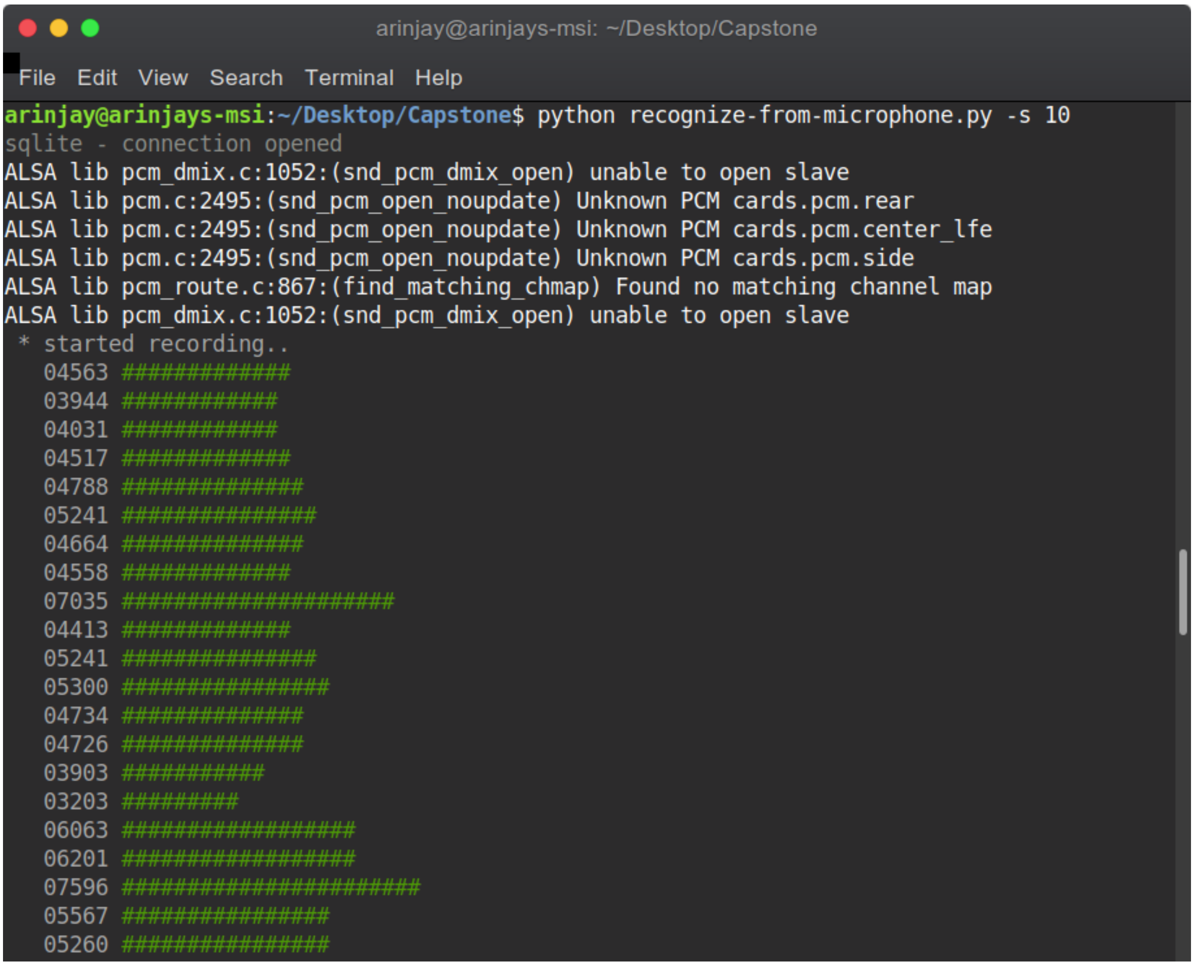
```
arinjay@arinjays-msi: ~/Desktop/Capstone
File Edit View Search Terminal Help
   07498 ######################
   02055 ######
   00639 #
   02756 ########
   03945 ############
   03733 ###########
   03291 ##########
   03475 ##########
   04052 ############
   04370 #############
   03777 ###########
   04092 ############
 * recording has been stopped
 * recorded 409600 samples
   fingerprinting channel 1/2
   local_maxima: 127 of frequency & time pairs
   ** found 1384 hash matches (step 1000/1673)
   ** found 866 hash matches (step 673/1673)
   finished channel 1/2, got 2250 hashes
   fingerprinting channel 2/2
   local_maxima: 123 of frequency & time pairs
   ** found 1419 hash matches (step 1000/1616)
   ** found 938 hash matches (step 616/1616)
   finished channel 2/2, got 4607 hashes

 ** totally found 4607 hash matches
 => song: Bohemian Rhapsody.mp3 (id=177)
    offset: 475 (22 secs)
    confidence: 453
sqlite - connection has been closed
```

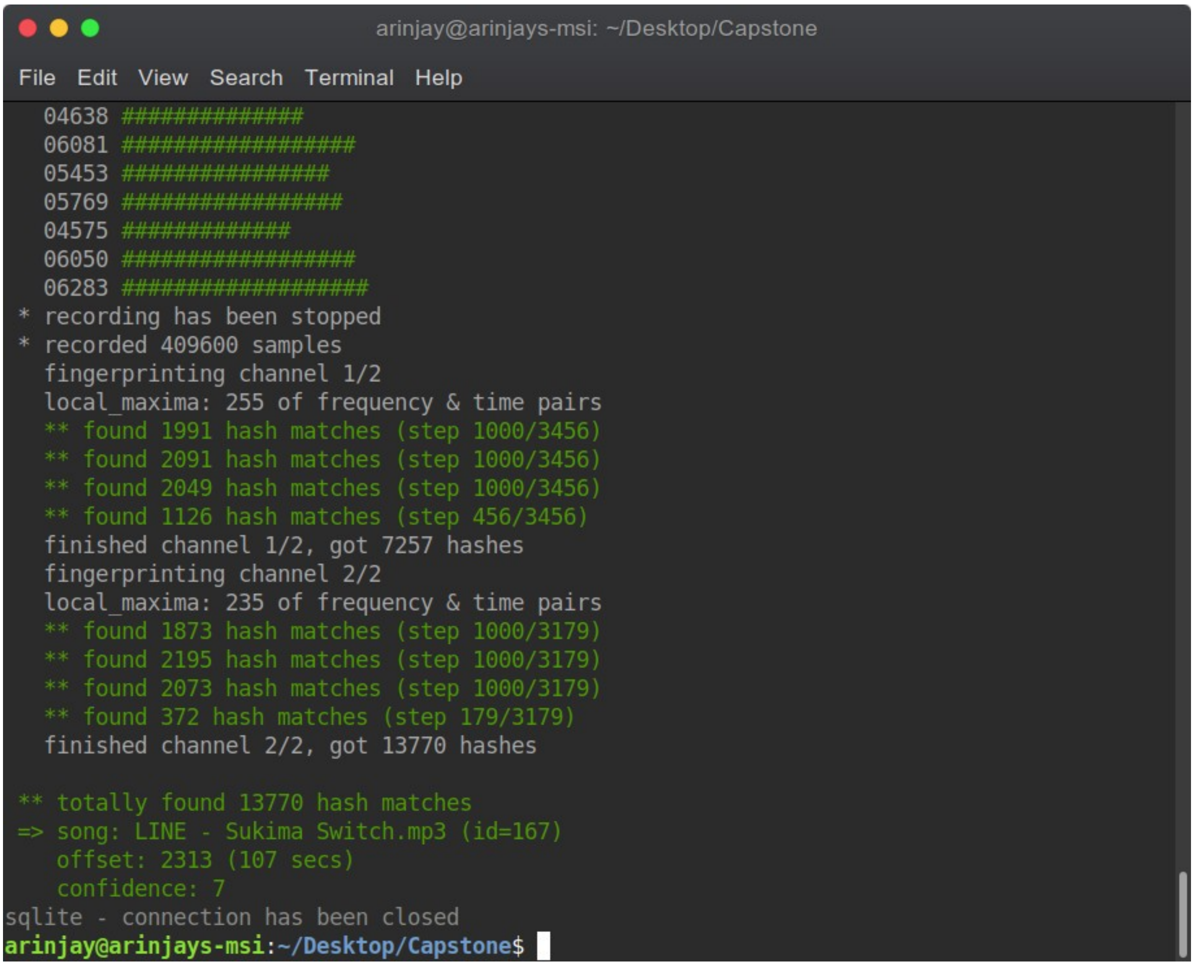
At the end of the recording, the song is fingerprinted in both stereo and mono channels (channel 1 and 2).

Then the hash values are matched for both channels and after the offset difference is calculated, the correct match from the database is then found and returned, along with the confidence level or the probability of it being a correct output and not a false positive.

A false positive is an output that gives us an "incorrect match" instead of saying that the input audio file does not have the correct match stored in the database. An example would be as follows:



The song being given as audio input is not actually stored in our database. It does not have any fingerprint or hash values stored, neither is it stored in the song table.
The program still gives us a "match" with a VERY low confidence value. Such a low confidence value only occurs in the case of a false positive case.



The false match occurs as some amplitude peaks might be similar in the two songs, but with a confidence of 7, it is obvious that the match is not a correct output and is in fact a false positive.

7. RESULT AND DISCUSSION

I wanted to find a relation between the the speed of retrieval of the match from the database and compare it with 2 of the most important deciding factors that would influence this:

1. Size of the Database: I noticed that with a small database, of around 10 songs fingerprinted and stored, the retrieval was always quite fast as fewer comparisons had to be made, but with the size of the database going into the order of a few gigabytes, the speed of getting a match had increased as well.

2. Length of the audio recorded: The next thing I tested was the confidence and speed of retrieval of a song when recorded for different times. The result can be viewed in the table below for different times, all for a 1.2GB database with 177 songs. The song I tested was Bohemian Rhapsody, always from the 10 second mark

TABLE 3: Confidence and Time for Match

Time audio file recorded	Confidence	Time for match
1	4 – False positive match	19.42 seconds
2	10 – correct match	22.62 seconds
3	29 – correct match	24.18 seconds
4	74 – correct match	26.61 seconds
5	122 – correct match	50 seconds
10	453 - correct match	74.45 seconds
15	662 – correct match	101.53 seconds
20	967 – correct match	136.87 seconds

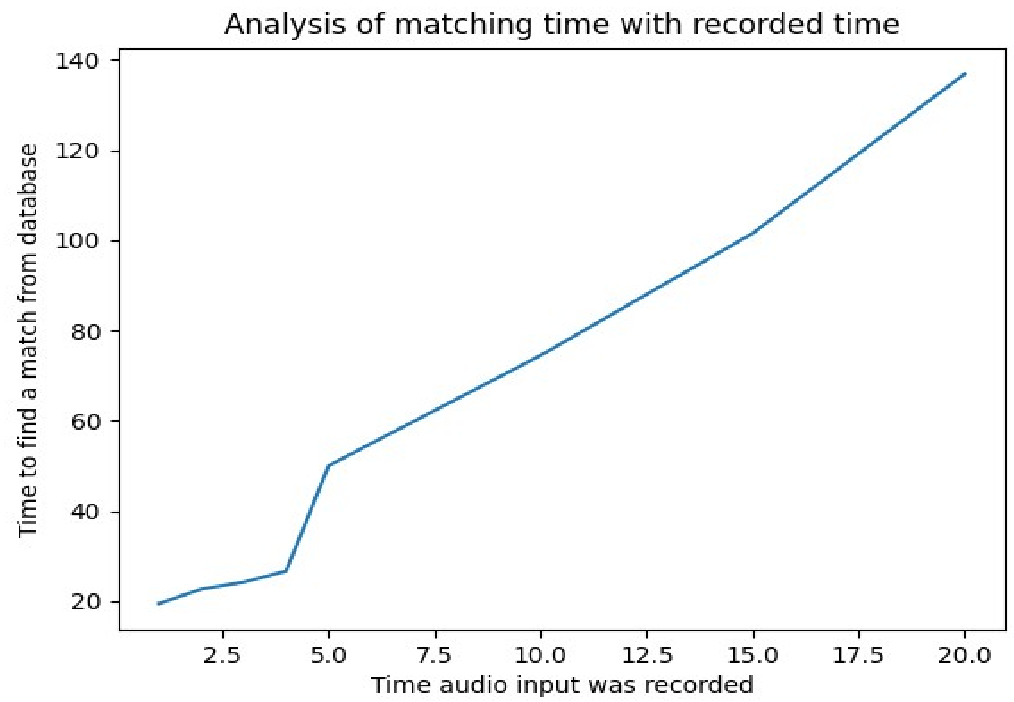


FIG 6: Analysis of matching time with recorded time

The above graph can be plotted using the values obtained for time analysis.

A graph can similarly be drawn between time of recorded audio file and the confidence of the match from the database:

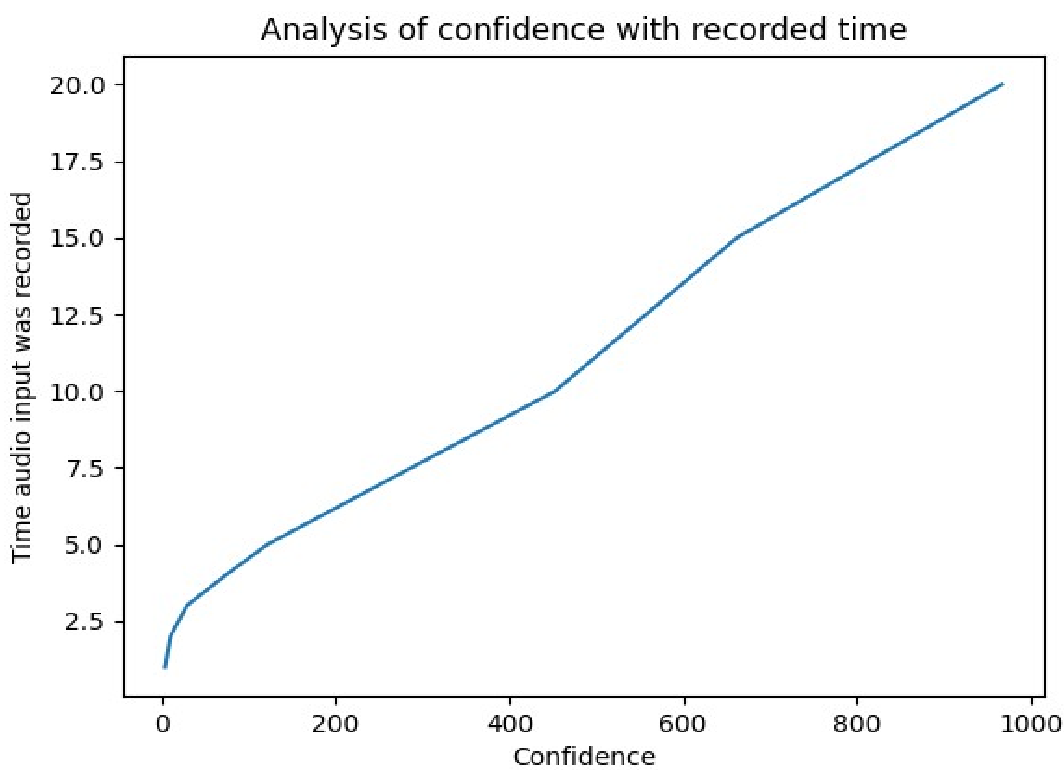


FIG 7: Analysis of confidence with recorded time

After running the same test on around 45 songs I tried to find the accuracy of finding a correct match against the recorded time of the audio file.

TABLE 4: Accuracy of Matches

Time recorded	Matches	Accuracy
1	2/10	20%
2	4/10	40%
3	7/10	70%
4	9/10	90%
5	10/10	100%

From this it is possible to say that a minimum of 5 seconds is needed to get accurate matches against the database with this implementation.

On plotting the graph

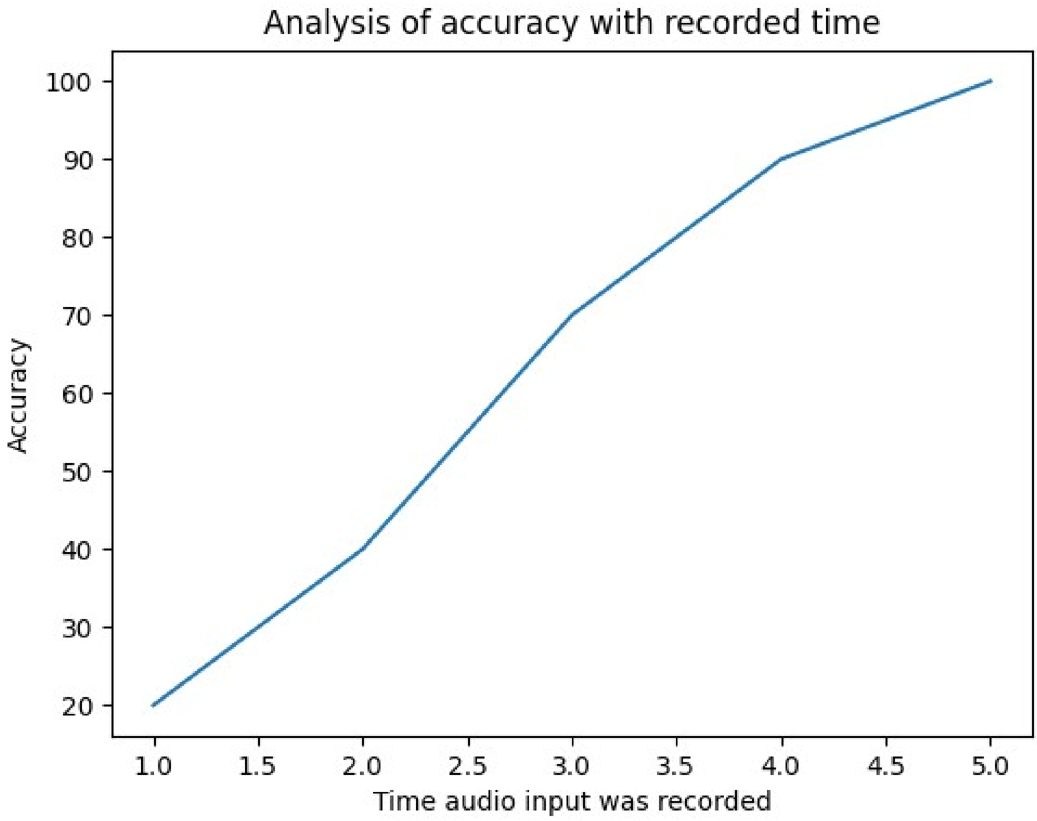


FIG 8: Analysis of accuracy with recorded time

References

[1] Hui Li Tan, Yongwei Zhu, Lekha Chaisorn, Haibin Huang (2009), Sequential Rhythmic Information Retrieval for Audio Similarity Matching, *TENCON 2009 - 2009 IEEE Region 10 Conference*

[2] Ya-Dong Wu, Yang Li, Bao-Long Liu (2003), A New Method for Approximate Melody Matching, *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)*

[3] Sazia Parvin, Jong Sou Park (2007), An Efficient Music Retrieval Using Noise Cancellation, *Future Generation Communication and Networking (FGCN 2007)*

[4] Zanchun Gao, Yuting Liu, Yanjun Jiang (2015), An effective method on Content Based Music Feature Extraction, *2015 IEEE Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*

[5] Takahiro Hayashi, Nobuaki Ishii, Masato Yamaguchi, Fast Music Information Retrieval with Indirect Matching, *2014 22nd European Signal Processing Conference (EUSIPCO)*

[6] Li, W., Zhang, X. & Wang, Z. Music content authentication based on beat segmentation and fuzzy classification. *J AUDIO SPEECH MUSIC PROC. 2013, 11 (2013).*

[7] F. Kurth and M. Muller, "Efficient Index-Based Audio Matching," in *IEEE Transactions on Audio, Speech, and Language Processing, vol. 16, no. 2, pp. 382-395, Feb. 2008.*

[8] W. Feng, N. Guan and Z. Luo, "High-performance audio matching with features learned by convolutional deep belief network," *2016 IEEE 13th International Conference on Signal Processing (ICSP), Chengdu, 2016, pp. 1724-1728.*

[9] E. R. Swedia, A. B. Mutiara, M. Subali and Ernastuti, "Deep Learning Long-Short Term Memory (LSTM) for Indonesian Speech Digit Recognition using LPC and MFCC Feature," 2018 Third International Conference on Informatics and Computing (ICIC), Palembang, Indonesia, 2018, pp. 1-5.

[10] Wang, Avery, "An Industrial Strength Audio Search Algorithm" 2003

[11] Palmirotta, Guendalina. (2016). A study of Shazam's Audio Recognition. 10.13140/RG.2.2.21768.21766.

[12] Erling Wold, Thom Blum, Douglas Keislar, James Wheaton, "Content-Based Classification, Search, and Retrieval of Audio", in IEEE Multimedia, Vol. 3, No. 3: FALL 1996, pp. 27-36