Protecting Copyright Ownership via Identification of Remastered Music in Radio Broadcasts

M. A. P. P. Marasinghe

Index No : 15000877

Supervisor : Dr. K.L. Jayaratne

Co-supervisor : Dr. M.I.E. Wickramasinghe

February 2020

Submitted in partial fulfillment of the requirements of the
B.Sc in Computer Science Final Year Project (SCS4124)

Protecting Copyright Ownership via Identification of Remastered Music in Radio Broadcasts

M.A.P.P. Marasinghe

Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name : M.A.P.P. Marasinghe

.....

Signature of Candidate Date :

This is to certify that this dissertation is based on the work of

Mr. M.A.P.P. Marasinghe

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor Name : Dr. K.L. Jayaratne

.....

Signature of Supervisor Date :

Co-supervisor Name : Dr. M.I.E. Wickramasinghe

.....

Signature of Co-supervisor Date :

Abstract

Music identification in radio broadcasts have different use cases like playlist generation and protecting copyright ownership of artistes. In real world environments, music is often remastered by radio channels to fit in to limited air times. Mostly those remastering includes time stretching, pitch shifting and etc. Therefore robustness of the music identification method plays a crucial role.

In this dissertation, we propose to use Scale Invariant Feature Transform (SIFT) on Short Time Fourier Transformation (STFT) spectrogram to extract audio descriptors. Experiments show that SIFT descriptors exhibit robustness against audio distortions such as time stretching and pitch shifting. Finally a ratio based threshold is used to differentiate identified and non-identified states.

Preface

The new method of remastered music identification method introduced in this research is a method refined by me in conjunction with my supervisor and co-supervisor. The song registration module, database structure and audio matching module are my own work. The results of the Section 5.2 rely upon experiments conducted by me.

Acknowledgement

I would like to express my very great appreciation to Dr. K.L. Jayaratne, my research supervisor for his valuable guidance, support and encouragement throughout this research work.

I would also like to give special thanks to my co-supervisor Dr. M.I.E. Wickramasinghe for his valuable and constructive suggestions during the planning and development of this research work and for his patient guidance and assistance in keeping my progress on schedule. Also his willingness to give his time so generously has been very much appreciated.

My special thanks are extended to the staff of Outstanding Song Creators Association (OSCA) for their assistance with the collection of my data.

I wish to acknowledge the help provided by my examiners Prof. G.K.A. Dias and Dr. M.G.N.A.S. Fernando for their valuable feedback in every evaluation to help me improve my work.

Finally, I wish to thank my beloved parents for their support, encouragement and being there for me in all my hardships.

Table of Contents

List of Figures

List of Tables

List of Acronyms

BLOB Binary Large Object

DoG Difference of Gaussians

DP Dynamic Programming

DTW Dynamic Time Warping

FN False Negative

FP False Positive

MIR Music Information Retrival

OSCA Outstanding Song Creators Association

OTI Optimal Transposition Index

PCA Principle Component Analysis

PCP Pitch Class Profiles

SIFT Scale Invariant Feature Transform

STFT Short Time Fourier Transformation

SVD Singular Value Decomposition

TN True Negative

TP True Positive

Chapter 1

Introduction

1.1 Background to the Research

According to the intellectual property act of Sri Lanka[2], royalties must be paid to the original artistes when a song is broadcast on a radio channel. Each radio channel is maintaining a playlist to keep track of the songs that were broadcast throughout the day. That playlist can later be used to pay royalties to the respective artistes. However, in order to streamline and regulate the royalty payment process, it is vital to have a method to monitor the radio broadcasts. Manual radio broadcast monitoring is infeasible and expensive due to increasing number of both radio channels and songs. In manual monitoring a person should be assigned to each channel who needs to keep record of each song in the radio broadcast of that assigned channel. Due to the increasing number of songs and the fallible nature of humans such a monitoring task is prone to errors and inaccuracies. Hence an automated radio broadcast monitoring approach must be considered as a viable alternative in the modern day radio broadcast monitoring.

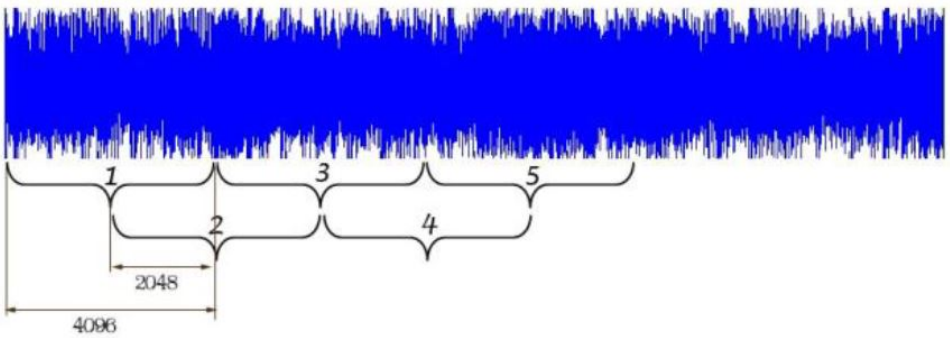

Figure 1.1: Key controlling parameters of STFT[1]

In the research "Radio Broadcast Monitoring to Ensure Copyright

Ownership"[1], researchers have implemented an automated radio broadcast monitoring system (refer the Figure 1.2 for the architecture) which has achieved 97.14% overall accuracy in identifying original songs in radio broadcasts. The researchers introduced an audio fingerprint to register and identify songs. The fingerprint was introduced as a series of hash values extracted from frequency domain audio signal. Time domain signals were converted to frequency domain by using STFT, which used 4096 bits long window and 2048 bits long overlapping area as shown in Figure 1.1. Then five peak values were extracted for each window by dividing mid frequency level into five bins and taking the peak value from each bin. Extracted five peak values were used to create a hash value as depicted in Figure 1.3.

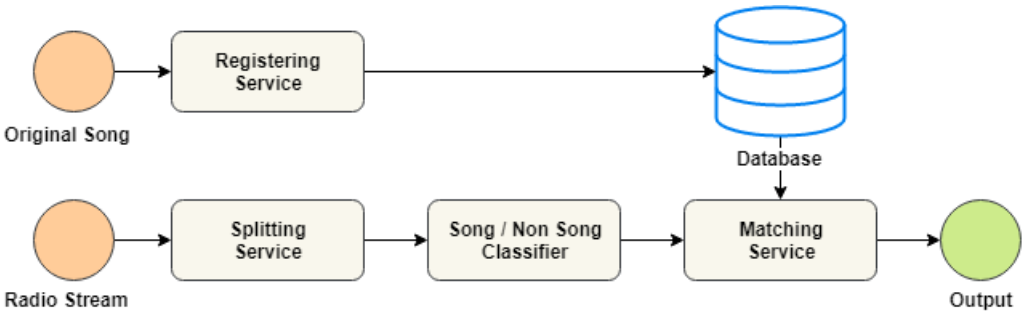

Figure 1.2: Architecture of the existing system

In contemporary radio broadcasts, channels tends to alter songs by including commercials and dialogues and by remastering the original song. Remastering can be done by adding or subtracting elements, or by changing pitch, equalization, dynamics or tempo[3]. Even though the above mentioned radio broadcast monitoring system's accuracy is not significantly affected by commercials and dialogues included in songs, the system is unable to identify a song when that song is remastered by the radio channel as changing pitch, equalization, dynamics or tempo directly affects both time domain and frequency domain audio data.

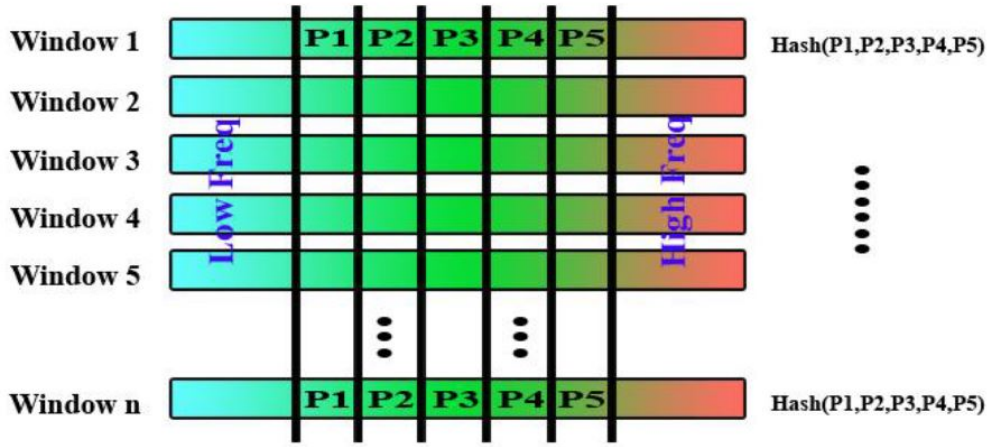

Figure 1.3: Extracting peaks and generating a hash value[1]

Timbre, tempo, timing, structure, key, harmonization and lyrics are the basic musical facets that can be identified[3]. Timbre, also known as tone colour is the music facet which makes a difference of different sound productions even when they have the same pitch and loudness. Simply it is what makes a difference between a piano and a violin playing the same note at the same volume. Timbre can be changed due to the use of different sound enhancing and processing techniques or to the use of different instruments and configurations. Tempo is the speed or pace of the music which can be easily changed by playing the music at different speeds. The music facet of timing is a rhythmic structure of the music which can be altered by the changes to the drum section. Structure is the arrangement of music sections, and music structure alterations can be made while remastering. Key, harmonization and lyrics are tonality, chords and words of the music which can be altered while remastering.

In order to identify remastered music in radio broadcasts, existing literature on cover song identification and music similarity measures can be used as foundation study to this research. Directly implementing a cover song identification method or a music similarity measure to identify remastered music in radio broadcasts is not possible as there is limited time to do the identification, because it is not just comparing two music clips to find similarity, but comparing a radio broadcast with a more than twenty thousand song database.

1.2 Research Problem and Research Questions

1.2.1 Project Aim

Aim of this research is to utilize computational theories and tools to protect copyright ownership of artistes in radio broadcasts.

1.2.2 Research Questions

Main three research questions are identified to address the challenges in music identification when remastered songs are broadcasted in radio.

1. What are forms of alterations to the basic musical facets in remastered music?
2. What are the approaches of identifying remastered music?
3. What approach can be used to identify remastered music in radio broadcasts?

1.2.3 Objectives

Answers to above research questions are obtained by accomplishing the five objectives.

1. Gather and identify the alterations made to remastered music when compared with the original music.
2. Review existing cover song identification methods and music similarity measures to implement feature extraction methods for relevant features.
3. Identify similarity descriptive features with respect to the identified forms of alterations.
4. Introduce a new music similarity descriptor using identified features.
5. Use introduced music similarity descriptor to identify remastered music in radio broadcasts.

1.3 Methodology

In the proposed method of remastered song identification, various algorithms are used to extract the audio features, create audio descriptors and match against stored descriptors. Hence we have divided our remastered song identification process in to five steps.

1. Preprocessing
2. Feature Extracting
3. Descriptor Storing (Registering)
4. Matching
5. Postprocessing

Processes of the above steps will be discussed in the following subsections.

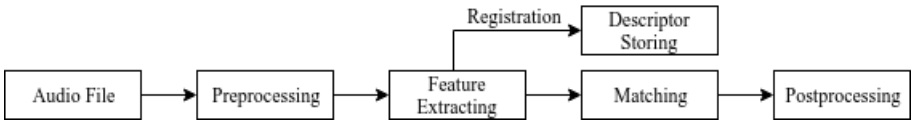

Figure 1.4: Remastered Song Identification Process

Preprocessing

In default audio data is represented in the time domain. Since even a small change in an audio changes the time domain representation drastically, using the time domain representation of the audio to extract features is not recommended. Hence time domain audio signal is converted to frequency domain signal by using STFT method. STFT is a sequence of Fourier transforms of a windowed signal[4]. A 2048 bits long window with 50% overlapping was used as STFT key parameters as depicted in Figure 1.5.

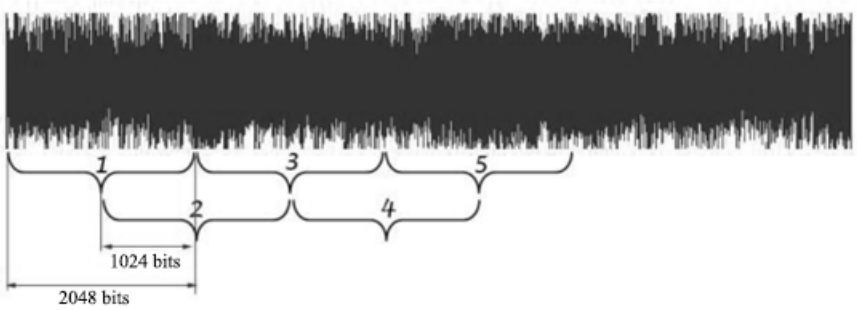

Figure 1.5: Key parameters on STFT. A 2048 bits long window with 1024 bits long overlapping area.

STFT is often visualized using its spectrogram[4], which is an intensity plot of STFT magnitude over time. The generated spectrogram is converted to a color image as shown in Figure 1.6. Axis labels and ticks are removed to stop identification of them as key points in feature extracting step.

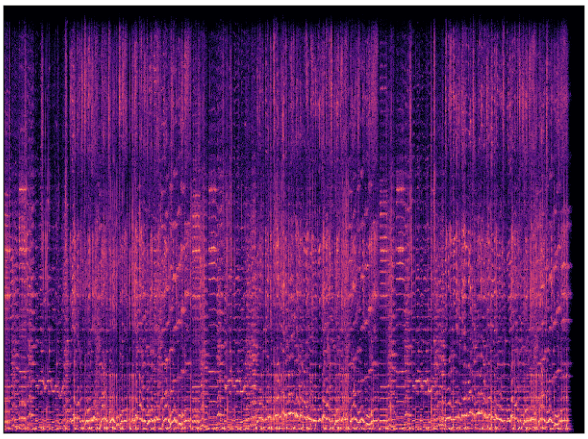

Figure 1.6: Generated colour image of spectrogram after preprocessing.

Feature Extracting

STFT spectrogram itself can be considered as an audio descriptor[5]. This method uses SIFT[6] to extract the features which are robust to music remastering. In Figure 1.7, it can be observed that when tempo is altered the spectrogram will either expand or compress with the time axis and when pitch is altered the spectrogram will either shift upwards or downwards with the frequency axis.

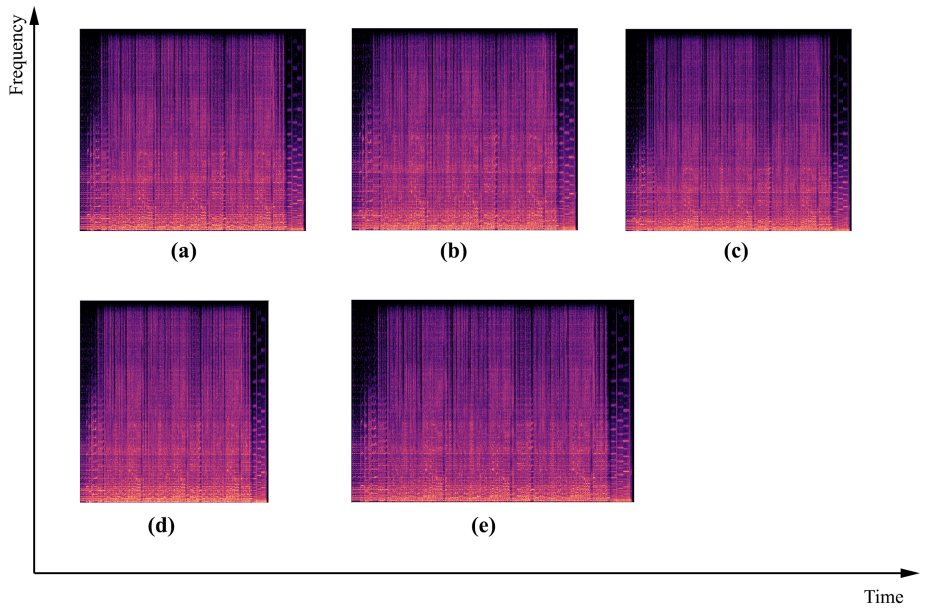

Figure 1.7: Spectrogram transformations on audio enhancements. **(a)** is the spectrogram image of an original song. **(b)** 20% pitch increase, **(c)** 20% pitch decrease, **(d)** 20% tempo increase and **(e)** 20% tempo decrease spectrogram images.

SIFT is used in computer vision to identify scale invariant features of an image. SIFT features are invariant to image rotation, scale alterations and illumination[6]. The SIFT feature extractor used in this method consists of four main steps.

1. Scale space extrema detection: Gaussian filters of different scales are applied to the image and potential key points are selected as local minima or maxima of the Difference of Gaussians (DoG) for multiple scales.

2. Keypoint localization: Keypoints that have low contrast or those that are poorly located along edges are filtered out.

3. Orientation assignment: One or more orientations are assigned to each keypoint based on local image gradient.

4. Keypoint descriptor generation: Orientation histograms are created for 4 x 4 pixel neighborhoods for each keypoint. Each histogram consists of 8 bins, hence a (4 x 4 x 8) 128 dimensional descriptor is generated.

A set of extracted 128 dimensional descriptors works together in describing the input audio file. Extracted SIFT features are invariant to image stretch and translation which makes them better features to be used in audio identification algorithm which is robust to tempo alterations and pitch shifting.

Descriptor Storing (Registering)

SIFT descriptors of original songs must be stored to use them in the matching step of the remastered song identification process. Generally 3-5 minute music clip will have around 2000 key points in its STFT spectrogram. Hence a 2000×128 matrix will be generated for each original song that will be registered.

The descriptor matrix of each original song is converted to a binary string and that binary string is stored in the database as Binary Large Object (BLOB)s. Converting to binary string and storing the matrix as a BLOB will ensure fast recreation of the matrix while retrieving[7].

Matching

Music identification is facilitated by matching a feature matrix of a query audio clip with a feature matrix of a original song. Final goal of the matching is to identify the count of keypoints that are matched with the original song. Identification of matching keypoints achieved by taking 2 nearest keypoints to a query keypoint and checking whether the distance to the closest keypoint is lesser than the $0.75 \times$ distance to the 2nd closest keypoint.

Postprocessing

The most similar song and matched keypoint count for a given query audio clip is identified in the matching step. But it doesn't exactly mean that query audio clip contains that song. Because the number of keypoints that were matched represents how much the query song matched to the most similar song. Hence there should be a threshold keypoint count to determine whether a query audio clip contains a song in our database or not. But using just a threshold value won't work here since different query audio clips generate different number of key points to match against the database. Hence ratio based threshold is recommended as a measure to determine whether the matched song is actually a correct match. Keypoint ratio can be obtained by the below equation.

$$\text{Keypoint ratio} = \frac{\text{Matched keypoint count}}{\text{Keypoints generated for query audio clip}}$$

Based on this keypoint ratio, a threshold is used to determine the validity of the match found. In order to find this threshold value we have used 844 different audio clips with variable durations to match against 2300 original songs. Those 844 audio clips had 519 audio clips which had songs and 325 audio clips which

didn't have songs from those 2300 original songs. And we calculated accuracy for 18 testcases which will be discussed in section 5.1, and took the average of those 18 accuracies for variable threshold values. Then results were illustrated as shown in the Figure 1.8.

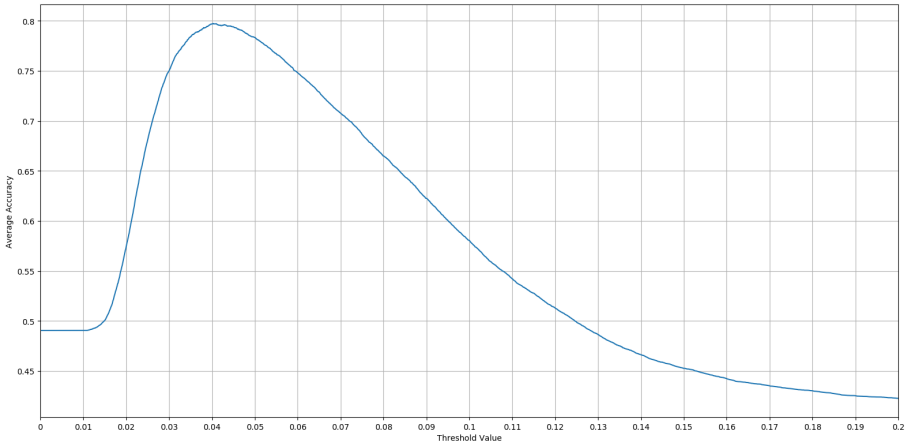

Figure 1.8: Average Accuracy Values for Different Threshold Values

Global peak can be observed in the illustration which makes that value a clear threshold point. 0.0403 is the threshold value that was found. Hence if keypoint ratio of a query audio is larger than 0.0403 then it's identified as a valid match to the song that was identified in the matching step, otherwise it's identified as a invalid match. This threshold point makes this method to clearly identify whether a query audio has a song which is in a database or not.

1.4 Outline of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 reviews the existing related works and summarizes them. Chapter 3 describes research design and methodology while Chapter 4 describes implementation details of the methodology. Experiments and their results are illustrated in Chapter 5. Chapter 6, the final chapter presents the conclusion, findings and future work.

1.5 Scope and Delimitations

1.5.1 In Scope

The following areas will be covered under the research project.

- Exploration of possible remastering techniques and outcomes.
- Introduction of music similarity descriptor.
- Identification of remastered music in radio broadcasts.

1.5.2 Out Scope

The following areas will not be covered under the research project.

- Identification of instrumental, acoustic or medley covers of original music.
- Identification of quotations in music such as lyrical quotations or musical quotations.

Chapter 2

Literature Review

2.1 Evolution of Audio

Music has an impact on every stage of life of a human being. Audio music has been growing with daily life styles since prehistoric times although there isn't a particular theory regarding whilst and wherein music originated. Historiographers state that there are six periods of music and each period has a specific genre of music that vastly contributed to the evaluation of music upto today. In the following subsections, each music period is briefly described.

2.1.1 Medieval Period (0-1400 A.D)

In this period, musical notations were developed which give us an understanding of how music sounded at that time. The monophonic and the polyphonic were the two commonly used types of music during this period and most of it were subjected to religion.

2.1.2 Renaissance Period (1400-1600 A.D)

Renaissance meaning "rebirth" states a period of a new era for music where creating and perceiving of music were rapidly changed. Music was composed according to the new idea of concerning the thoughts of the listener at that time. Most of the music artists could move the listeners to devotion by composing music for the church.

2.1.3 Baroque Period (1600-1750 A.D)

The word "baroque" means bizzare which comes from an Italian word "Barocco". The Baroque period is commonly known for complex pieces and intricate harmonies and the idea of the modern orchestra was created with the help of opera.

2.1.4 Classical Period (1750-1800 A.D.)

The Classical period expanded from the Baroque period making music less complex. Simpler melodies and forms such as the sonatas were used with the usage of the piano as the primary music instrument. In this era turning into public concerts is considered as a milestone in the history of music.

2.1.5 Romantic Period (1800 1900 A.D.)

Music expanded from Classical period by adding expressions and feelings with the use of various musical instruments such as wind instruments. Melodies were more focused on gravitating drama and emotions.

2.1.6 Contemporary Period (1900-Present)

Over time music has moved on to a complete free reign. Nowadays composers try to do new things by ultimate experimentation and the usage of technology. Music has come a long way and it has a long way to go and we are grateful for the countless composers who have contributed to what music is today.

2.2 Audio File Formats

Every type of digital data needs a file format to be stored in a computer system and the file format for digital audio data is audio file format. File format should be taken into consideration since audio data could vary from different formats in signal processing. Audio file formats can be categorized into three major groups.

2.2.1 Uncompressed Audio Formats

Uncompressed audio format is considered as the main type of audio format which takes a larger space to be stored but preserves the quality of the sound. It is vastly used on professional audio workstations such as the television and film industry. Eg: WAV, AIFF, CDA

2.2.2 Lossless Compressed Audio Formats

Compared to the above type, lossless compressed audio formats take less space to be stored. It maintains a compression ratio of about 2:1 and reduces processing time. Eg: FLAC, Wav Pack, Monkey's audio, ALAC

2.2.3 Lossy Compressed Audio Formats

This type of audio formats reduce the size of the audio file which results in reducing the quality of sound as well. But compared to the loss of data and quality, this file format is used highly in limited storage capacities. Eg: Mp3, AAC

2.3 Remastered Audio Identification

Remastered song identification falls under the domain of cover song identification, which is a very active area of study in the Music Information Retrival (MIR) community [3]. Literature on cover song identification contains different approaches taken to measure and model music similarity in both symbolic and audio domains. Literature relevant to cover song identification can be divided into few areas such as query-by-humming systems, content-based music retrieval, genre classification and audio fingerprinting.

In symbolic domain of cover song identification, symbolic representations of musical content is used in content processing. Query-by-humming systems [8] fall under the symbolic domain as in query-by-humming systems music contents are stored and processed in symbolic representations. This query-by-humming method is parallel to retrieving cover songs from a song database. Even though techniques used in query-by-humming systems could be useful in future approaches of cover song identification, these systems can't achieve high accuracy on real world audio music signals [9, 3].

Audio domain cover song identification approaches focus on measuring similarity of music by exploiting music facets shared between two songs. Extracting invariant features is used to exploit shared music facets. Although such extracted descriptors are responsible to overcome majority of facet changes, special stress is given for achieving tempo, key and structure since those facets are not usually managed by the extracted descriptors themselves [3]. Hence we can look at existing literature in terms of feature extraction, tempo invariance, key invariance, structure invariance and finally similarity comparison. Furthermore, we can take approaches which fall into this general pipeline (refer Figure 2.1) to look at different techniques used for these stages and distinguish each approach from one another by those techniques.

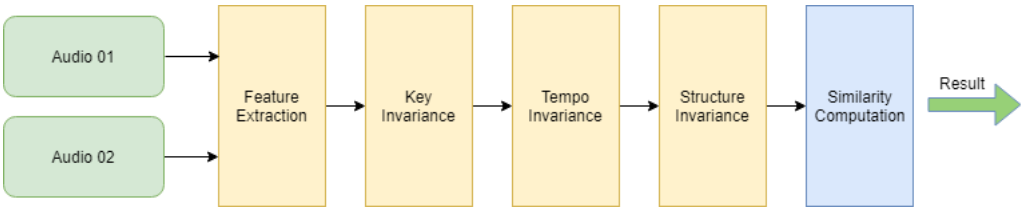

Figure 2.1: General Pipeline for Cover Song Identification

Bello's cover song identification method extracts chord sequences as the feature and uses K transpositions for key invariance. Even though there is no technique used for structure invariance, Dynamic Programming (DP) is used for tempo invariance. Finally it uses edit distance to compute the similarity [10]. Since there is no technique used for structure invariance, that method is inefficient against the structural changes in cover songs. Egorov proposed another method which uses the same general pipeline with extracting Pitch Class Profiles (PCP) as the feature. But in this method Egorov uses Optimal Transposition Index (OTI) for key invariance and DP is used for both tempo and structural invariance. And match length is used for similarity computation [11].

Foote [12] and Izmirli [13] introduced two methods which were using Dynamic Time Warping (DTW) for similarity computation and DP for tempo invariance. Both the methods lack techniques for key invariance and structure invariance which makes those methods to perform inefficient in both key and temporal changes. The feature extracted by Foote is energy spectrum while Izmirli extracted key templates. Marlot uses the same techniques for tempo invariance and similarity computation which are DP and DTW, but melody is the extracted feature which uses the key estimation for key invariance.

Research	Feature	Key Invariance	Tempo Invariance	Structure Invariance	Similarity Computation
Bello [10]	**Chords**	**K transpositions**	**DP**	**-**	**Edit distance**
Egorov & Linetsky [11]	**PCP**	**OTI**	**DP**	**DP**	**Match length**
Foote [12]	**Energy spectral**	**-**	**DP**	**-**	**DTW**
Izmiril [13]	**Key templates**	**-**	**DP**	**-**	**DTW**
Marolt [14]	**Melody**	**Key estimation**	**DP**	**-**	**DTW**

Table 2.1: Cover song identification methods and their techniques used for each step in general pipeline

Related works mentioned above for audio domain can be modelled to the general pipeline for cover song identification (refer Figure 2.1) as described in Table 2.1.

Chapter 3

Design

3.1 Architectural Design

The basic framework to identify music in radio broadcasts is proposed in "Radio Broadcast Monitoring to Ensure Copyright Ownership"[1]. And it's currently implemented and deployed in Outstanding Song Creators Association (OSCA). Hence the infrastructure to contain the basic framework of a radio monitoring system is already there. When different methods of music identification are applied to radio broadcast monitoring, only registering service, database and matching service will be changed conserving the other modules that are in the architecture as shown in the Figure 3.1.

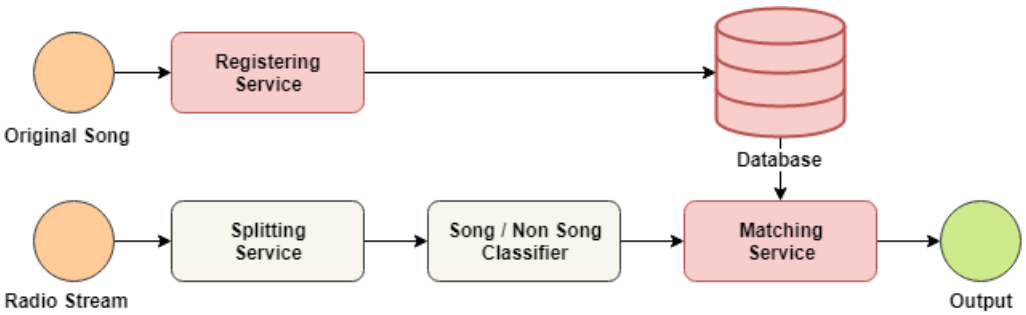

Figure 3.1: Architectural Design

Registering service takes original songs as input and generates specific descriptors to be stored in the database. Database stores the generated audio descriptors by the registering service in a retrieval friendly framework to help the matching service to retrieve descriptors faster. The matching service takes a query audio clip and determines whether that query audio clip contains a song registered before.

3.2 Principle Component Analysis (PCA)

There are many different approaches to identify music by extracting different audio features as discussed in the Chapter 2. Since there are very low number of researches conducted on sinhala music identification, finding audio features which can differentiate two sinhala songs was required to continue the research. Hence Principle Component Analysis (PCA) was conducted to find features on a 5000 song dataset extracting 27 different audio features and results were collected for different normalization techniques. Singular Value Decomposition (SVD) was used to composite multi-dimension features.

3.2.1 PCA with Raw Dataset

Initially a PCA was executed on the raw feature matrix without any normalization technique which led to the results in Figure 3.2. Abnormality of the result caused to revisit the feature matrix and need of a normalization technique was identified.

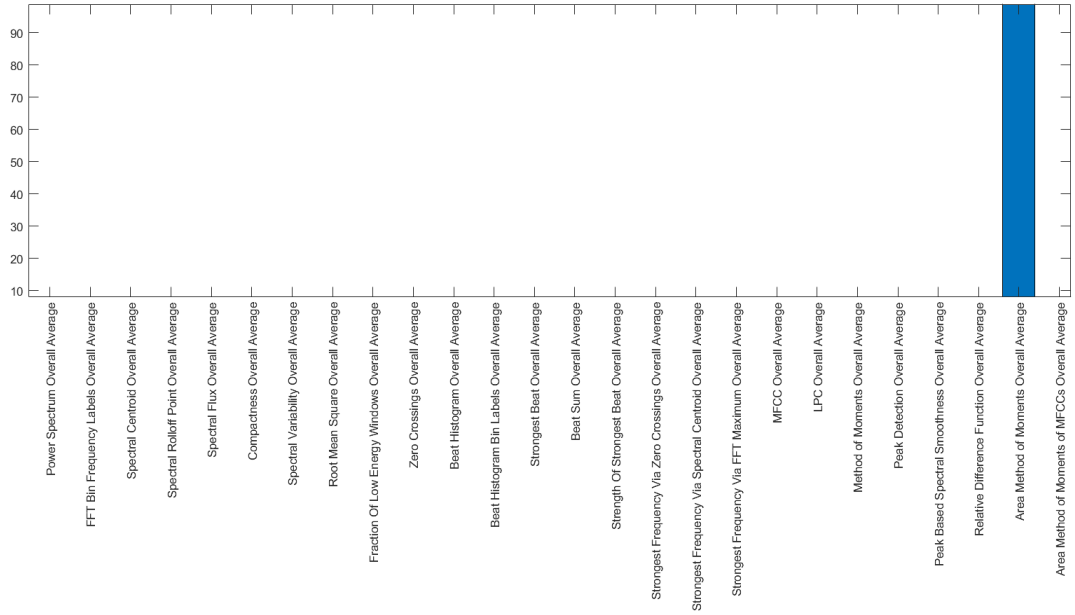

Figure 3.2: PCA coefficients weighted by eigen values

3.2.2 PCA with Dataset Normalized by Z-score

Then PCA was conducted on the dataset which was normalized by z-score which produced results on Figure 3.3. While analyzing the results, it was observed that results indicate uniform variance among the features that was not present in the original feature matrix. Hence need of a normalization technique which preserve the variance of a feature was identified.

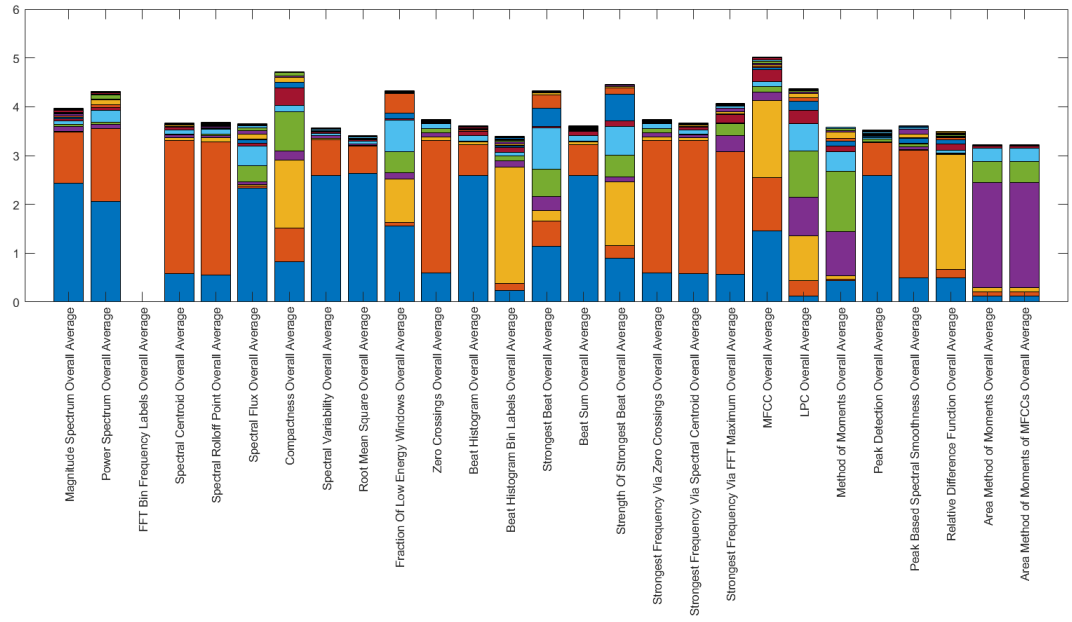

Figure 3.3: PCA coefficients weighted by eigen values (Normalized by Zscore)

3.2.3 PCA with Dataset Normalized by Rescaling

Finally the dataset was normalized by rescaling and the conducted PCA gave a justifiable result which is shown in Figure 3.3. "Area Method of Moments" and "Area Method of Moments of MFCCs" are identified as the two features which covers majority of the variance in the dataset.

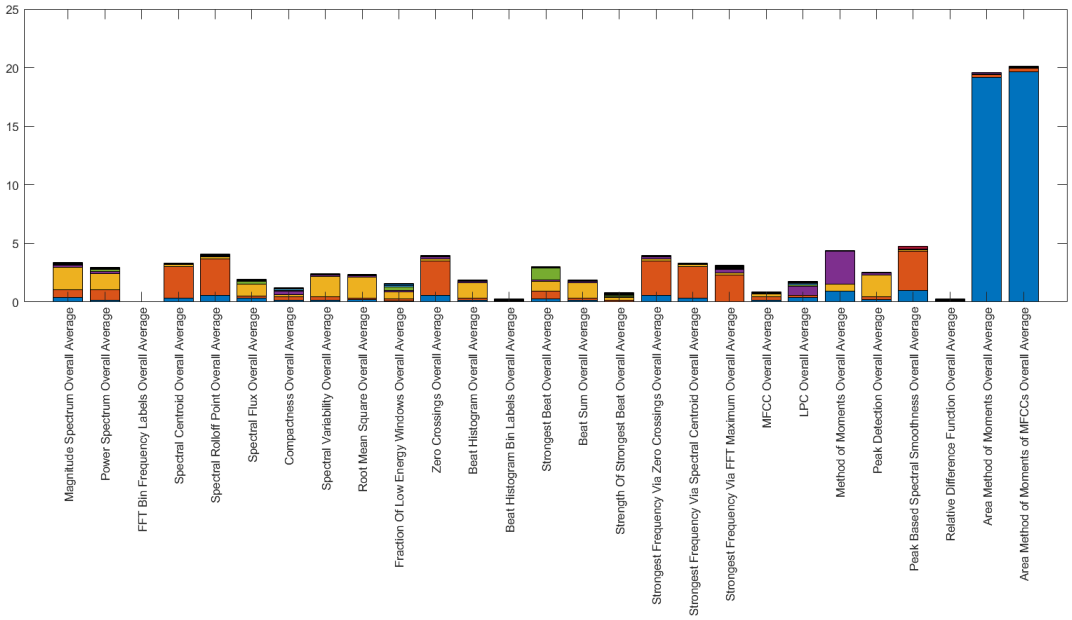

Figure 3.4: PCA coefficients weighted by eigen values (Normalized by Rescaling)

Then those two features were extracted from audio clips with slight changes to

tempo and pitch, in order to check whether those two features show any invariance to pitch or the tempo. But when tempo or pitch changed slightly, SVD values of those features changed drastically. Therefore results of the PCA couldn't be used to progress on this research which resulted to revisit the literature to find better features to extract.

3.3 Scale Invariant Feature Transform (SIFT) Based Approach

SIFT is a method of extracting features from a image which are invariant to scale modifications[6]. Approaching music identification with image processing techniques found in literature as discussed in Chapter 2. Pitch shifting can be identified as an expansion or a compression of the STFT spectrogram over the frequency axis, while tempo alterations can be identified as an expansion or a compression over the time axis as depicted in Figure 3.5. Hence SIFT can be used to find similarity of a tempo altered or pitch altered audio using it's STFT spectrogram.

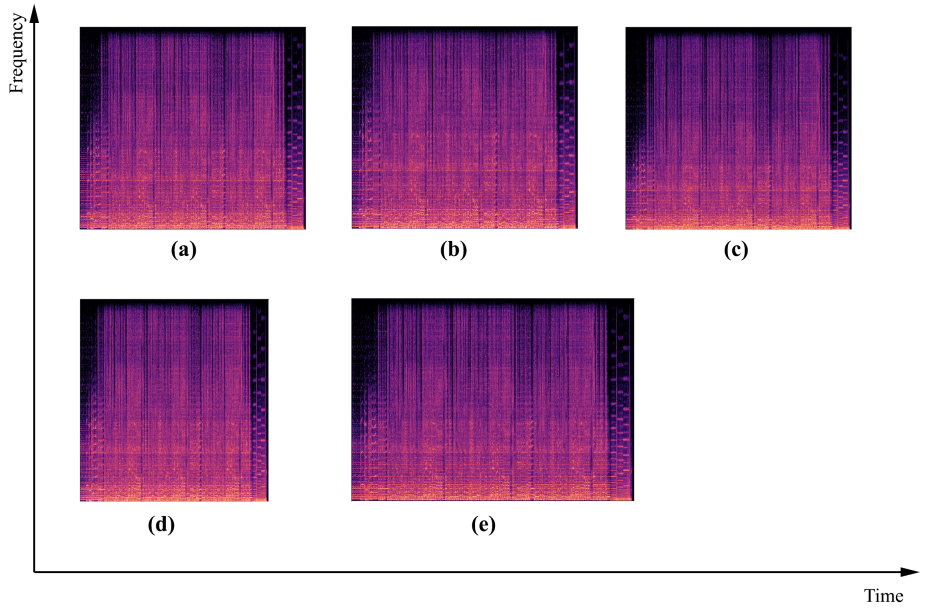

Figure 3.5: Spectrogram transformations on audio enhancements. **(a)** is the spectrogram image of a original song. **(b)** 20% pitch increase, **(c)** 20% pitch decrease, **(d)** 20% tempo increase and **(e)** 20% tempo decrease spectrogram images.

3.3.1 Short Time Fourier Transform

STFT[4] is used to transform a signal from time domain to frequency domain as time domain signals are unstable. When selecting key parameters 2048 bits long

window was used with 50% overlap as depicted in Figure 3.6, in order to ensure that every part of a signal is represented in two windows.

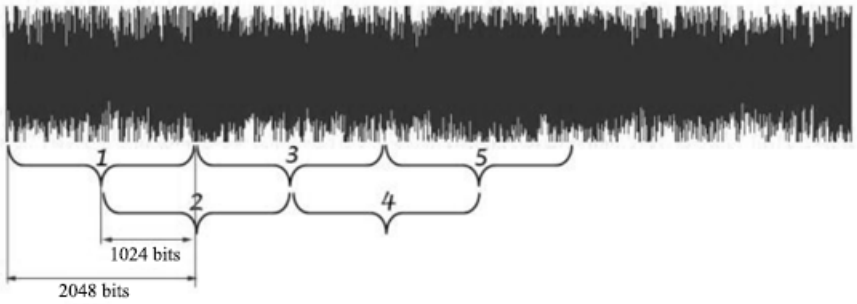

Figure 3.6: Key parameters on STFT. 2048 bits long window with 1024 bits long overlapping area.

STFT is often visualized using its spectrogram[4], which is an intensity plot of STFT magnitude over time. The generated spectrogram is converted to a color image as shown in Figure 3.7. Axis labels and ticks are removed to stop identification of them as key points in feature extracting step.

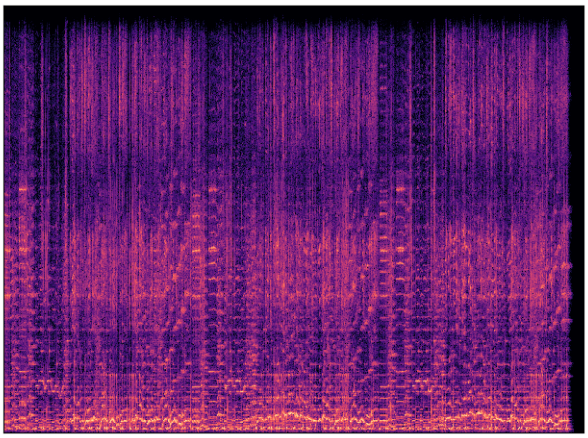

Figure 3.7: Generated colour image of a spectrogram

3.3.2 SIFT Descriptor Extraction

SIFT is used in computer vision to identify scale invariant features of an image. SIFT features are invariant to image rotation, scale alterations and illumination[6]. The SIFT feature extractor used in this method consists of four main steps.

1. Scale space extrema detection: Gaussian filters of different scales are applied to the image and potential key points are selected as local minima or maxima of the DoG for multiple scales.

2. Keypoint localization: Keypoints that have low contrast or those that are poorly located along edges are filtered out.

3. Orientation assignment: One or more orientations are assigned to each keypoint based on local image gradient.

4. Keypoint descriptor generation: Orientation histograms are created for 4 x 4 pixel neighborhoods for each keypoint. Each histogram consists 8 bins, hence (4 x 4 x 8) 128 dimensional descriptor is generated.

SIFT generate (No of Keypoints $\times$ 128) dimension matrix for each spectrogram. In average, more than 1000 keypoints are identified from a single spectrogram as shown in the Figure 3.8.

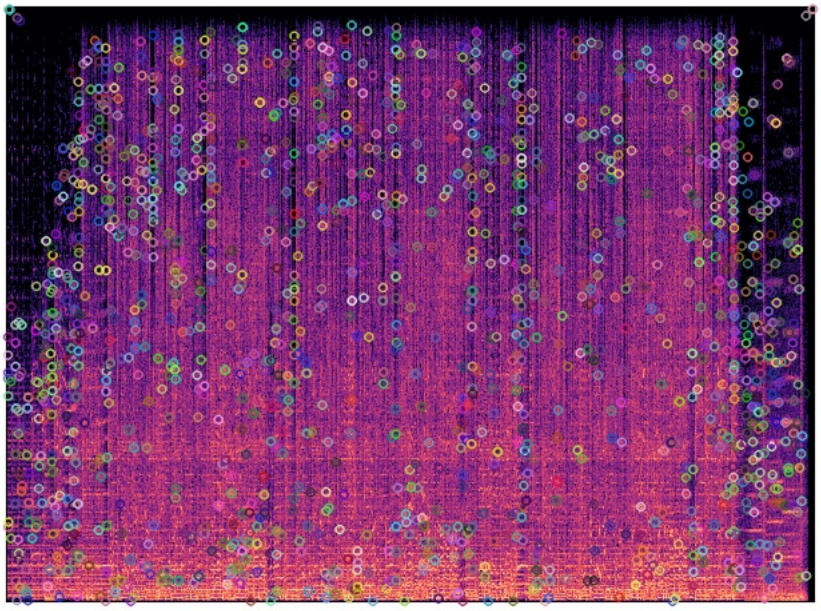

Figure 3.8: Identified keypoints in a SIFT spectrogram

3.3.3 SIFT Descriptor Matching

Music identification is facilitated by matching a feature matrix of a query audio clip with a feature matrix of a original song. Final goal of the matching is to identify the count of keypoints that are matched with the original song. Identification of matching keypoints achieved by taking 2 nearest keypoints to a query keypoint and checking whether the distance to the closest keypoint is lesser than the 0.75 $\times$ distance to the 2nd closest keypoint. Those keypoints which were matched can be visualized as shown in the Figure 3.9.

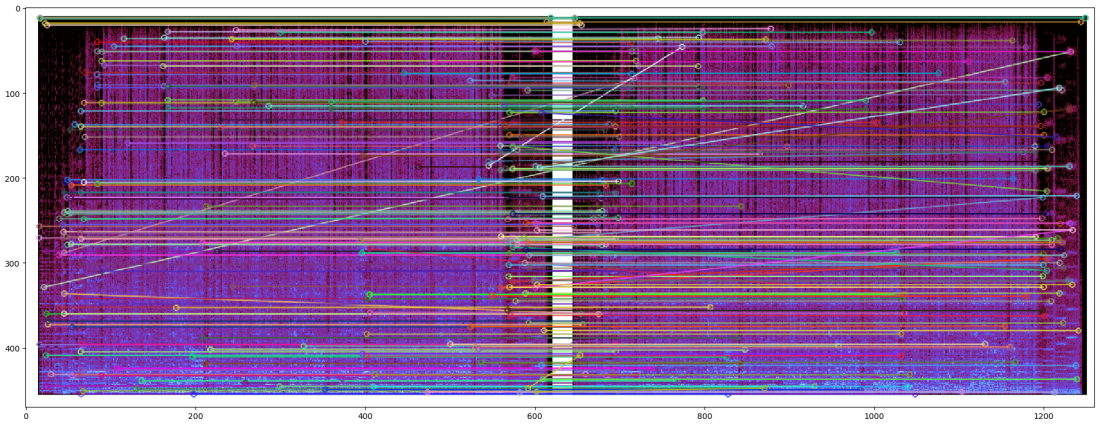

Figure 3.9: Matched keypoints of two spectrograms

3.3.4 Thresholding

Final step is to determine whether that query audio clip has the matched original song or not. It's archived by a thresholding method. In the begging keypoint count was used as the threshold measure. But it had drawbacks such as when query audio is variable length number of keypoints identified will be different which eventually leads to different number of keypoints matched. Hence just using keypoint count as a threshold measure didn't work as finding a generalized threshold value is really different.

Therefore a ratio based threshold measure was recommended. Where we consider both number of keypoints that matched and initially identified keypoints count. Keypoint ratio can be obtained by below equation.

$$\text{Keypoint ratio} = \frac{\text{Matched keypoint count}}{\text{Keypoints generated for query audio clip}}$$

Keypoint ratio can be used as better threshold measure as we can find generalized threshold value irrespective of length of the query audio clip. Threshold value selection and results will be discussed on Section 5.2.

Chapter 4

Implementation

4.1 Principle Component Analysis (PCA)

4.1.1 Dataset

Song dataset with 5000 songs were obtained from the OSCA as mp3 files with 320kbps quality. Then those 5000 songs were converted to wav format with 16-pcm quality by using ffmpeg library.

4.1.2 Features and Tools

Jaudio library[15] was used to extract 27 different audio features from the song dataset. Then SVD function in Matlab[16] was used to composite multi-dimension features to a single value. Finally PCA implementation in Matlab[16] used to run the PCA.

4.2 Scale Invariant Feature Transform (SIFT) Based Approach

4.2.1 Short Time Fourier Transform

STFT implementation of librosa[17] library used to get the STFT spectrogram of the audio signal. Parameters given to off both axis bars on generated STFT spectrogram as spectrogram needed to be isolated.

4.2.2 SIFT Descriptor Extraction and Matching

OpenCV[18] SIFT implementation was used to identify keypoints and generate SIFT descriptors. Brute-force matcher implementation on OpenCV[18] was the matcher used to match two SIFT feature vectors.

Chapter 5

Results and Evaluation

5.1 Experiments

5.1.1 Song Dataset

A song dataset consisting 2300 sinhala songs were used in the registration step of the experiment. These 2300 sinhala songs were retrieved from OSCA of Sri Lanka which works as the governing organization to ensure intellectual property rights of music in Sri Lanka.

5.1.2 Query Audio Samples

Variable sized 844 query audio clips were used for the experiment to evaluate the performance of this method against different durations. 519 of the above mentioned audio clips had songs which are in the database while 325 audio clips didn't have songs from the database. Hence for each test case, the sample size was 844 query audio clips with 0.61492 prevalence.

5.1.3 Test Cases

Test cases were created by doing audio distortions to the query audio samples. Performance of the method was evaluated for three main audio distortions which are tempo alteration, pitch alteration and both pitch and tempo alteration. Both increased and decreased alterations are considered for three different levels of alterations which are 10% alteration, 20% alteration and 50% alteration. Hence there are 3 audio distortions, 2 audio distortion directions and 3 audio alteration levels, 18 ($3 \times 2 \times 3$) test cases were generated to evaluate the performance.

5.2 Experiment Results

The proposed method has an exact way to identify whether a query audio clip has a matching registered song or not. Therefore this method can be considered as a classifier. A classifier can be evaluated by the confusion matrix generated for a given sample. True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN) were calculated for each test case. Then accuracy and FP rate was calculated for each test case. Reducing FP rate is significant as much as increasing the accuracy given that this method mainly focuses on identifying music on radio broadcasts. Accuracy and FP rate can be calculated from the formulas given below.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{FP Rate} = \frac{FP}{FP + TN}$$

Keypoint ratio based threshold is proposed as the threshold measure in this method. Two experiments were done, one with keypoint count as threshold measure and the other with keypoint ratio as threshold measure to validate the claim of keypoint ratio is better threshold measure.

5.2.1 Using Keypoint Count as Threshold

Average accuracies for different threshold values are illustrated as shown in Figure 5.1 to identify the optimal threshold value to use. Since the visualization clearly indicates a global peak value which is keypoint count of 69, that value can be used as the threshold value. This threshold value of 69 is used in this experiment.

Results of the experiment using keypoint count as the threshold measure are presented in Table 5.1. There is no clear change between accuracies of pitch changes and tempo changes when using keypoint count as the threshold measure. Pitch changes and tempo changes up to 20% alteration can be identified with 92%-98% accuracy.

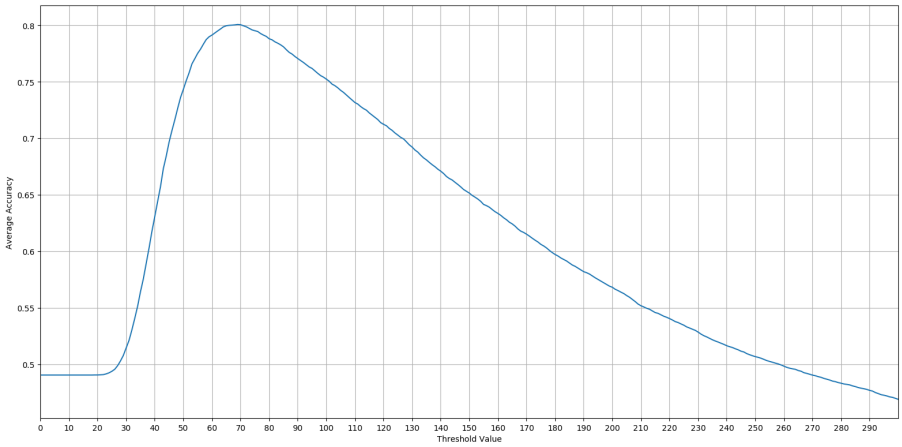

Figure 5.1: Average Accuracy Values for Different Threshold Values (Keypoint Count)

Test Case	**TP**	**FP**	**TN**	**FN**	**Accuracy**	**FP Rate**
Tempo Increase 10%	509	19	306	10	0.96564	0.05846
Tempo Increase 20%	509	18	307	10	0.96682	0.05538
Tempo Increase 50%	497	10	315	22	0.96209	0.03077
Tempo Decrease 10%	515	16	309	4	0.97630	0.04923
Tempo Decrease 20%	516	19	306	3	0.97393	0.05846
Tempo Decrease 50%	517	22	303	2	0.97156	0.06769
Pitch Increase 10%	514	21	304	5	0.96919	0.06462
Pitch Increase 20%	513	11	314	6	0.97986	0.03385
Pitch Increase 50%	7	27	298	512	0.36137	0.08308
Pitch Decrease 10%	511	11	314	8	0.97749	0.03385
Pitch Decrease 20%	463	4	321	56	0.92891	0.01231
Pitch Decrease 50%	0	0	325	519	0.38507	0.00000
Tempo & Pitch Increase 10%	414	11	314	105	0.86256	0.03385
Tempo & Pitch Increase 20%	339	18	307	180	0.76540	0.05538
Tempo & Pitch Increase 50%	0	26	299	519	0.35427	0.08000
Tempo & Pitch Decrease 10%	373	5	320	146	0.82109	0.01538
Tempo & Pitch Decrease 20%	226	2	323	293	0.65047	0.00615
Tempo & Pitch Decrease 50%	0	0	325	519	0.38507	0.00000

Table 5.1: Experiment results using keypoint count as threshold

5.2.2 Using Keypoint Ratio as Threshold

Average accuracies for different threshold values are illustrated as shown in Figure 5.2 to identify the optimal threshold value to use. Since the visualization clearly indicates a global peak value which is keypoint count of 0.0403, that value can be used as the threshold value. This threshold value of 0.0403 is used in this experiment.

Results of the experiment using keypoint ratio as the threshold measure presented in Table 5.2. It can be observed that accuracies have increased considerably compared to results of the experiment which uses keypoint count as the threshold measure which validates the claim of taking keypoint ratio as a better threshold measure. Pitch changes and tempo changes up to 20% alteration can be identified with 95%-99% accuracy.

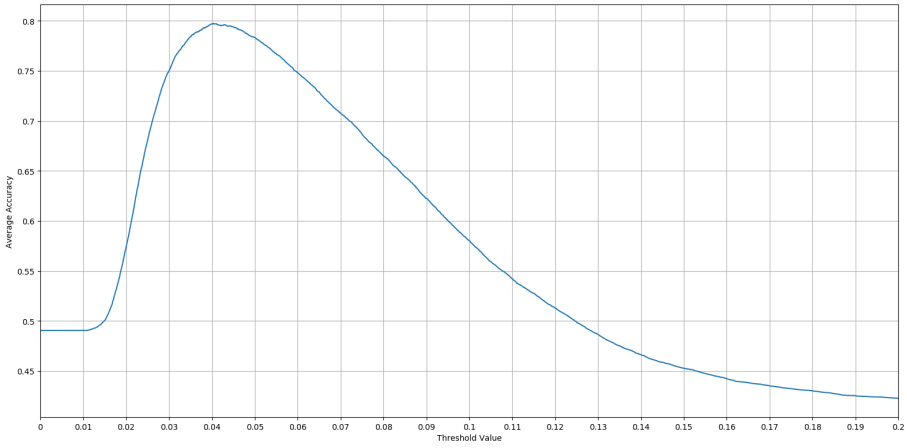

Figure 5.2: Average Accuracy Values for Different Threshold Values (Keypoint Ratio)

Test Case	**TP**	**FP**	**TN**	**FN**	**Accuracy**	**FP Rate**
Tempo Increase 10%	512	13	312	7	0.97630	0.04000
Tempo Increase 20%	508	10	315	11	0.97512	0.03077
Tempo Increase 50%	501	9	316	18	0.96801	0.02769
Tempo Decrease 10%	515	11	314	4	0.98223	0.03385
Tempo Decrease 20%	519	10	315	0	0.98815	0.03077
Tempo Decrease 50%	519	9	316	0	0.98934	0.02769
Pitch Increase 10%	519	9	316	0	0.98934	0.02769
Pitch Increase 20%	517	9	316	2	0.98697	0.02769
Pitch Increase 50%	0	4	321	519	0.38033	0.01231
Pitch Decrease 10%	516	16	309	3	0.97749	0.04923
Pitch Decrease 20%	503	25	300	16	0.95142	0.07692
Pitch Decrease 50%	23	160	165	496	0.22275	0.49231
Tempo & Pitch Increase 10%	413	11	314	106	0.86137	0.03385
Tempo & Pitch Increase 20%	287	17	308	232	0.70498	0.05231
Tempo & Pitch Increase 50%	0	7	318	519	0.37678	0.02154
Tempo & Pitch Decrease 10%	432	28	297	87	0.86374	0.08615
Tempo & Pitch Decrease 20%	346	34	291	173	0.75474	0.10462
Tempo & Pitch Decrease 50%	6	154	171	513	0.20972	0.47385

Table 5.2: Experiment results using keypoint ratio as threshold

5.3 Evaluation

Final results of the experiment with keypoint ratio as the threshold measure performs significantly better than the keypoint count threshold measure. Hence, it can be concluded that keypoint ratio is a better threshold measure than keypoint count.

There is a sudden drop of accuracies between 20% and 50% pitch changes. This sudden drop happens when shifting up or down on the frequency axis making spectrogram local patterns to be compressed or expanded which makes the pattern to get unidentified as shown in Figure 5.3.

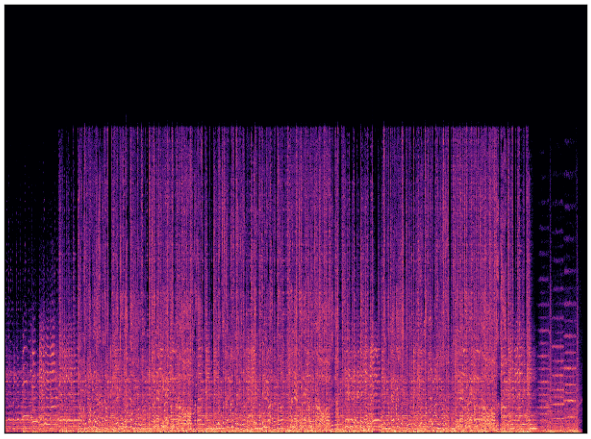

Figure 5.3: Generated colour image of a spectrogram (50% pitch decrease)

Chapter 6

Conclusions

A review on the accomplishment of research objectives to answer research questions, limitations of current work and the directions for future enhancements are included in this chapter.

6.1 Conclusion on Research Questions

Alterations to tempo, key and noise audio facets are made to music by remastering as discussed in Chapter 3. Music with noise alterations can be identified with "Radio Broadcast Monitoring to Ensure Copyright Ownership"[1]. Therefore, identifying music with tempo and key alterations was crucial to identify remastered music in radio broadcasts.

There are several approaches to identify music with tempo and key alterations as introduced in Chapter 2. Majority of the literature to identify music with tempo and key alterations were STFT based. Using SIFT descriptors to match two STFT spectrograms suits identification of remastered songs as it can be used to match variable sized audio clips with original songs. Above 97% accuracy can be achieved for tempo alterations up to 20% and above 95% accuracy can be achieved for pitch alterations up to 20% as discussed on Section 5.2. Hence, it can be concluded that research questions were successfully answered by accomplishing research objectives.

6.2 Limitations

In the proposed remastered song identification method, the matching step involves retrieving N number of $M \times 128$ dimension matrix where N is the number of registered songs. That retrieval can be identified as the bottle neck of the system as data retrieval from the database is time wise inefficient. Efficiency of the whole system depends on the data retrieval speed from the database.

6.3 Auxiliary Findings

6.3.1 Common Features of False Positives

Analysis of FPs is useful to have an insight about the fingerprinting method. Therefore STFT spectrograms and audio clips of FPs were analyzed and found that those audio clips contain rapid key translations compared to an average sinhala song.

6.3.2 Impact on SIFT Descriptor by Tempo Decrease

It has been observed that accuracy on tempo decreased audio identification has more accuracy than identification of original songs itself. Hence we can assume that tempo decreasing enhances the invariant feature which is extracted by the SIFT descriptor on STFT spectrogram.

6.4 Future Works

This research can be enhanced to identify remastered songs with more audio alterations which are not covered by this research such as music additions and music subtractions. And the matching process can be improved to reduce search space and reduce the number of calculations.

Proposed method of this research is implemented as an isolated system, but it should be implemented with the other parts of the "Radio Broadcast Monitoring to Ensure Copyright Ownership"[1], so this method can be used to identify remastered songs in radio broadcasts.

References

[1] E. D. N. W. Senevirathna and K. L. Jayaratne, "Radio Broadcast Monitoring to Ensure Copyright Ownership," *International Journal on Advances in ICT for Emerging Regions (ICTer)*, vol. 11, p. 1, Aug. 2018.

[2] Parliament of the democratic socialist republic of Sri Lanka, "Intellectual Property Act, No.36 of 2003."

[3] J. Serrà, E. Gómez, and P. Herrera, "Audio Cover Song Identification and Similarity: Background, Approaches, Evaluation, and Beyond," in *Advances in Music Information Retrieval* (J. Kacprzyk, Z. W. Raś, and A. A. Wieczorkowska, eds.), vol. 274, pp. 307–332, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[4] N. Kehtarnavaz, "Frequency Domain Processing," *Digital Signal Processing System Design*, vol. 1, pp. 175–196, 2008.

[5] Y. Ke, D. Hoiem, and R. Sukthankar, "Computer vision for music identification," *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*, vol. I, pp. 597–604, 2005.

[6] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[7] R. Sears, C. Van Ingen, and J. Gray, "To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?," tech. rep., 2006.

[8] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith, "Query by humming: musical information retrieval in an audio database," in *Proceedings of the third ACM international conference on Multimedia - MULTIMEDIA '95*, (San Francisco, California, United States), pp. 231–236, ACM Press, 1995.

[9] R. B. Dannenberg, W. P. Birmingham, B. Pardo, N. Hu, C. Meek, and G. Tzanetakis, "A comparative evaluation of search techniques for query-by-humming using the MUSART testbed," *Journal of the American*

Society for Information Science and Technology, vol. 58, pp. 687–701, Mar. 2007.

[10] J. P. Bello, "Audio-Based Cover Song Retrieval Using Approximate Chord Sequences: Testing Shifts, Gaps, Swaps and Beats.," in *ISMIR*, vol. 7, pp. 239–244, Citeseer, 2007.

[11] A. Egorov and G. Linetsky, "Cover song identification with IF-F0 pitch class profiles," *MIREX extended abstract*, 2008.

[12] J. Foote, "ARTHUR: Retrieving Orchestral Music by Long-Term Structure.," in *ISMIR*, 2000.

[13] Özgür İzmirli, "Tonal Similarity from Audio Using a Template Based Attractor Model.," in *ISMIR*, pp. 540–545, Citeseer, 2005.

[14] M. Marolt, "A Mid-level Melody-based Representation for Calculating Audio Similarity.," in *ISMIR*, pp. 280–285, 2006.

[15] D. McEnnis, C. McKay, I. Fujinaga, and P. Depalle, "JAUDIO: A FEATURE EXTRACTION LIBRARY," p. 5.

[16] MATLAB, *R2018b*. Natick, Massachusetts: The MathWorks Inc., 2018.

[17] "LibROSA — librosa 0.7.0 documentation," *https://librosa.github.io/librosa/*.

[18] "OpenCV," *https://opencv.org/*.

Appendices

Appendix A

Codebase of the Implementation

A.1 Code File Structure

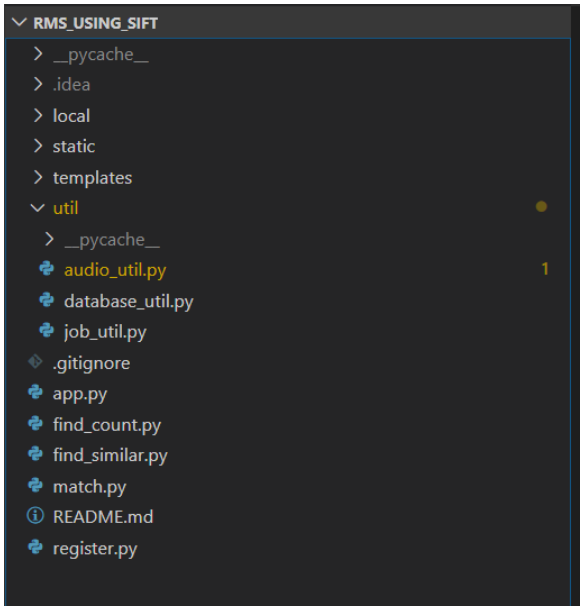

Figure A.1: File structure of the codebase

A.2 File Contents

A.2.1 register.py

```
from util.job_util import Job

if __name__ == '__main__':
    register_job = Job(Job.type["register"])
    print("Job Started. Job ID : " + str(register_job.job_id))
    register_job.start_job(None)
```

A.2.2 match.py

```
from util.job_util import Job

if __name__ == '__main__':
```

```
    match_job = Job(Job.type["match"])
    print("Job Started. Job ID : " + str(match_job.job_id))
    match_job.start_job(None)
```

A.2.3 audio-util.py

```
import librosa
import librosa.display
import numpy as np
import matplotlib.pyplot as plt
import io
import cv2 as cv
from util.database_util import DatabaseUtil
from multiprocessing import Pool
import os.path
import gc
import datetime

class AudioUtil:

    @staticmethod
    def find_count(job_id):
        gc.enable()
        db_util = DatabaseUtil()
        song_list = db_util.execute_query("SELECT id FROM song where id<'10
        for (query_id,) in song_list:
            for des_count in range(19):
                des_name = "des_"+str(des_count)
                des_node = db_util.execute_query("SELECT "+des_name+" FROM
                for (query_descriptor,) in des_node:
                    query_descriptor = np.frombuffer(query_descriptor, dtyp
                    query_descriptor = query_descriptor.reshape(-1, 128)
                    keypoint_count = query_descriptor.shape[0]
                    print(str(query_id)+" : "+str(query_id)+" : "+str(keypo
                    db_util.execute_update("INSERT INTO keypoint_count VALU
        db_util.execute_update("UPDATE job SET status='3' WHERE id='" + str
        db_util.close()
        gc.collect()

    @staticmethod
    def match_songs(job_id):
        gc.enable()

        db_util = DatabaseUtil()

        song_ids = db_util.execute_query("SELECT id FROM song where id > '1
        for (song_id,) in song_ids:
            AudioUtil.match_song(song_id)
            print("Song : "+str(song_id)+" Done")
        db_util.execute_update("UPDATE job SET status='3' WHERE id='" + str
        db_util.close()
        gc.collect()

    @staticmethod
    def find_similar_songs(job_id):
        db_util = DatabaseUtil()
        bf = cv.BFMatcher()
        song_list = db_util.execute_query("SELECT id,des_0 FROM song")
        for (query_id, query_descriptor) in song_list:
            query_descriptor = np.frombuffer(query_descriptor, dtype=np.flo
            query_descriptor = query_descriptor.reshape(-1, 128)
            for (list_id, list_descriptor) in song_list:
                list_descriptor = np.frombuffer(list_descriptor, dtype=np.f
                list_descriptor = list_descriptor.reshape(-1, 128)
                matches = bf.knnMatch(list_descriptor, query_descriptor, k=
                count = 0
                for m, n in matches:
                    if m.distance < 0.75 * n.distance:
                        count += 1
                if (count > 500) and (query_id!=list_id):
                    db_util.execute_insert("INSERT INTO same_song (song_id,
```

```
        db_util.execute_update("UPDATE job SET status='3' WHERE id='" + str
        db_util.close()
        gc.collect()


    @staticmethod
    def match_song(song_id):
        db_util = DatabaseUtil()
        bf = cv.BFMatcher()
        song_list = db_util.execute_query("SELECT id,des_0 FROM song where
        descriptor_list = db_util.execute_query("SELECT * FROM song WHERE i
        for (id, title, des_0, des_1, des_2, des_3, des_4, des_5, des_6, de
            best_song = [0]*19
            best_count = [0]*19

            des_0 = np.frombuffer(des_0, dtype=np.float32)
            des_0 = des_0.reshape(-1, 128)

            des_1 = np.frombuffer(des_1, dtype=np.float32)
            des_1 = des_1.reshape(-1, 128)

            des_2 = np.frombuffer(des_2, dtype=np.float32)
            des_2 = des_2.reshape(-1, 128)

            des_3 = np.frombuffer(des_3, dtype=np.float32)
            des_3 = des_3.reshape(-1, 128)

            des_4 = np.frombuffer(des_4, dtype=np.float32)
            des_4 = des_4.reshape(-1, 128)

            des_5 = np.frombuffer(des_5, dtype=np.float32)
            des_5 = des_5.reshape(-1, 128)

            des_6 = np.frombuffer(des_6, dtype=np.float32)
            des_6 = des_6.reshape(-1, 128)

            des_7 = np.frombuffer(des_7, dtype=np.float32)
            des_7 = des_7.reshape(-1, 128)

            des_8 = np.frombuffer(des_8, dtype=np.float32)
            des_8 = des_8.reshape(-1, 128)

            des_9 = np.frombuffer(des_9, dtype=np.float32)
            des_9 = des_9.reshape(-1, 128)

            des_10 = np.frombuffer(des_10, dtype=np.float32)
            des_10 = des_10.reshape(-1, 128)

            des_11 = np.frombuffer(des_11, dtype=np.float32)
            des_11 = des_11.reshape(-1, 128)

            des_12 = np.frombuffer(des_12, dtype=np.float32)
            des_12 = des_12.reshape(-1, 128)

            des_13 = np.frombuffer(des_13, dtype=np.float32)
            des_13 = des_13.reshape(-1, 128)

            des_14 = np.frombuffer(des_14, dtype=np.float32)
            des_14 = des_14.reshape(-1, 128)

            des_15 = np.frombuffer(des_15, dtype=np.float32)
            des_15 = des_15.reshape(-1, 128)

            des_16 = np.frombuffer(des_16, dtype=np.float32)
            des_16 = des_16.reshape(-1, 128)

            des_17 = np.frombuffer(des_17, dtype=np.float32)
            des_17 = des_17.reshape(-1, 128)

            des_18 = np.frombuffer(des_18, dtype=np.float32)
            des_18 = des_18.reshape(-1, 128)

            for (query_id,query_descriptor) in song_list:

                query_descriptor = np.frombuffer(query_descriptor, dtype=np
                query_descriptor = query_descriptor.reshape(-1, 128)

                # Descriptor 0
                matches = bf.knnMatch(des_0, query_descriptor, k=2)
```

```
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[0]:
    best_count[0] = count
    best_song[0] = query_id

# Descriptor 1
matches = bf.knnMatch(des_1, query_descriptor, k=2)
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[1]:
    best_count[1] = count
    best_song[1] = query_id

# Descriptor 2
matches = bf.knnMatch(des_2, query_descriptor, k=2)
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[2]:
    best_count[2] = count
    best_song[2] = query_id

# Descriptor 3
matches = bf.knnMatch(des_3, query_descriptor, k=2)
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[3]:
    best_count[3] = count
    best_song[3] = query_id

# Descriptor 4
matches = bf.knnMatch(des_4, query_descriptor, k=2)
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[4]:
    best_count[4] = count
    best_song[4] = query_id

# Descriptor 5
matches = bf.knnMatch(des_5, query_descriptor, k=2)
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[5]:
    best_count[5] = count
    best_song[5] = query_id

# Descriptor 3
matches = bf.knnMatch(des_6, query_descriptor, k=2)
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[6]:
    best_count[6] = count
    best_song[6] = query_id

# Descriptor 3
matches = bf.knnMatch(des_7, query_descriptor, k=2)
count = 0
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        count += 1
if count > best_count[7]:
```

```
        best_count[7] = count
        best_song[7] = query_id

    # Descriptor 8
    matches = bf.knnMatch(des_8, query_descriptor, k=2)
    count = 0
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            count += 1
    if count > best_count[8]:
        best_count[8] = count
        best_song[8] = query_id

    # Descriptor 9
    matches = bf.knnMatch(des_9, query_descriptor, k=2)
    count = 0
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            count += 1
    if count > best_count[9]:
        best_count[9] = count
        best_song[9] = query_id

    # Descriptor 10
    matches = bf.knnMatch(des_10, query_descriptor, k=2)
    count = 0
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            count += 1
    if count > best_count[10]:
        best_count[10] = count
        best_song[10] = query_id

    # Descriptor 11
    matches = bf.knnMatch(des_11, query_descriptor, k=2)
    count = 0
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            count += 1
    if count > best_count[11]:
        best_count[11] = count
        best_song[11] = query_id

    # Descriptor 12
    matches = bf.knnMatch(des_12, query_descriptor, k=2)
    count = 0
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            count += 1
    if count > best_count[12]:
        best_count[12] = count
        best_song[12] = query_id

    # Descriptor 13
    matches = bf.knnMatch(des_13, query_descriptor, k=2)
    count = 0
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            count += 1
    if count > best_count[13]:
        best_count[13] = count
        best_song[13] = query_id

    # Descriptor 14
    matches = bf.knnMatch(des_14, query_descriptor, k=2)
    count = 0
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            count += 1
    if count > best_count[14]:
        best_count[14] = count
        best_song[14] = query_id

    # Descriptor 15
    matches = bf.knnMatch(des_15, query_descriptor, k=2)
```

```
                count = 0
                for m, n in matches:
                    if m.distance < 0.75 * n.distance:
                        count += 1
                if count > best_count[15]:
                    best_count[15] = count
                    best_song[15] = query_id

                # Descriptor 16
                matches = bf.knnMatch(des_16, query_descriptor, k=2)
                count = 0
                for m, n in matches:
                    if m.distance < 0.75 * n.distance:
                        count += 1
                if count > best_count[16]:
                    best_count[16] = count
                    best_song[16] = query_id

                # Descriptor 17
                matches = bf.knnMatch(des_17, query_descriptor, k=2)
                count = 0
                for m, n in matches:
                    if m.distance < 0.75 * n.distance:
                        count += 1
                if count > best_count[17]:
                    best_count[17] = count
                    best_song[17] = query_id

                # Descriptor 18
                matches = bf.knnMatch(des_18, query_descriptor, k=2)
                count = 0
                for m, n in matches:
                    if m.distance < 0.75 * n.distance:
                        count += 1
                if count > best_count[18]:
                    best_count[18] = count
                    best_song[18] = query_id

            for descriptor_id in range(19):
                db_util.execute_insert("INSERT INTO song_match (song_id, be
        db_util.close()
        gc.collect()

    @staticmethod
    def match_clip(job_id):
        clip_path = "/home/rms_sys_user/rms_using_sift/test/1667.mp3"
        date_time = datetime.datetime(2019, 10, 23, 0, 0, 0)
        db_util = DatabaseUtil()
        gc.enable()
        if os.path.isfile(clip_path):
            frame_id = db_util.execute_insert(
                "INSERT INTO frame_match (job_id, start, best_song, best_co
                    job_id) + "','" + date_time.strftime('%Y-%m-%d %H-%M-%S
            audio_stream, _ = librosa.load(clip_path)
            AudioUtil.match_frame(audio_stream, frame_id)
        db_util.execute_update("UPDATE job SET status='3' WHERE id='" + str
        db_util.close()

    @staticmethod
    def match_frame(audio_stream, frame_id):
        db_util = DatabaseUtil()
        song_list = db_util.execute_query("SELECT id,des_0 FROM song")
        clip_descriptor = AudioUtil.generate_descriptors(audio_stream)
        best_song = -1
        best_count = -1
        for (id, descriptor) in song_list:
            test_descriptor = np.frombuffer(descriptor, dtype=np.float32)
            test_descriptor = test_descriptor.reshape(-1, 128)
            bf = cv.BFMatcher()
            matches = bf.knnMatch(clip_descriptor, test_descriptor,  k=2)
            count = 0
            for m, n in matches:
                if m.distance < 0.75 * n.distance:
                    count += 1
```

```
            if count > best_count:
                best_count = count
                best_song = id
        db_util.execute_update("UPDATE frame_match SET best_song='" + str(b
            best_count) + "' where id='" + str(frame_id) + "'")
        db_util.close()
        gc.collect()

    @staticmethod
    def register_songs(job_id):
        gc.enable()
        thread_pool = Pool(24)
        db_util = DatabaseUtil()
        number_of_songs = db_util.execute_query("SELECT COUNT(id) FROM song
        number_of_songs = number_of_songs[0][0]
        file_directory = "/var/www/html/osca/storage/app/public/songs/"
        processes = []
        if number_of_songs == 0:
            existing_songs = DatabaseUtil.get_song_list()
            for (id, title) in existing_songs:
                thread = thread_pool.apply_async(AudioUtil.insert_song, (id
                processes.append(thread)
        for i in processes:
            i.get()
        thread_pool.close()
        thread_pool.join()
        db_util.execute_update("UPDATE job SET status='3' WHERE id='" + str

    @staticmethod
    def insert_song(id, title, file_directory):
        db_util = DatabaseUtil()
        song_file = file_directory + str(id) + ".mp3"
        if os.path.isfile(song_file):
            audio_stream, sample_rate = librosa.load(song_file)
            descriptor = AudioUtil.generate_descriptors(audio_stream)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("INSERT into song (id, title, des_0
                                        + "('" + str(id) + "','" + str(titl

            audio_fast_1 = librosa.effects.time_stretch(audio_stream, 1.1)
            descriptor = AudioUtil.generate_descriptors(audio_fast_1)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_1=%s WHERE id

            audio_fast_2 = librosa.effects.time_stretch(audio_stream, 1.2)
            descriptor = AudioUtil.generate_descriptors(audio_fast_2)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_2=%s WHERE id

            audio_fast_3 = librosa.effects.time_stretch(audio_stream, 1.5)
            descriptor = AudioUtil.generate_descriptors(audio_fast_3)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_3=%s WHERE id

            audio_slow_1 = librosa.effects.time_stretch(audio_stream, 0.9)
            descriptor = AudioUtil.generate_descriptors(audio_slow_1)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_4=%s WHERE id

            audio_slow_2 = librosa.effects.time_stretch(audio_stream, 0.8)
            descriptor = AudioUtil.generate_descriptors(audio_slow_2)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_5=%s WHERE id

            audio_slow_3 = librosa.effects.time_stretch(audio_stream, 0.5)
            descriptor = AudioUtil.generate_descriptors(audio_slow_3)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_6=%s WHERE id

            audio_fast_1 = librosa.effects.pitch_shift(audio_stream, sample
            descriptor = AudioUtil.generate_descriptors(audio_fast_1)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_7=%s WHERE id
```

```
            audio_fast_2 = librosa.effects.pitch_shift(audio_stream, sample
            descriptor = AudioUtil.generate_descriptors(audio_fast_2)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_8=%s WHERE id

            audio_fast_3 = librosa.effects.pitch_shift(audio_stream, sample
            descriptor = AudioUtil.generate_descriptors(audio_fast_3)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_9=%s WHERE id

            audio_slow_1 = librosa.effects.pitch_shift(audio_stream, sample
            descriptor = AudioUtil.generate_descriptors(audio_slow_1)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_10=%s WHERE id

            audio_slow_2 = librosa.effects.pitch_shift(audio_stream, sample
            descriptor = AudioUtil.generate_descriptors(audio_slow_2)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_11=%s WHERE id

            audio_slow_3 = librosa.effects.pitch_shift(audio_stream, sample
            descriptor = AudioUtil.generate_descriptors(audio_slow_3)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_12=%s WHERE id

            audio_fast_1 = librosa.effects.time_stretch(audio_fast_1, 1.1)
            descriptor = AudioUtil.generate_descriptors(audio_fast_1)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_13=%s WHERE id

            audio_fast_2 = librosa.effects.time_stretch(audio_fast_2, 1.2)
            descriptor = AudioUtil.generate_descriptors(audio_fast_2)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_14=%s WHERE id

            audio_fast_3 = librosa.effects.time_stretch(audio_fast_3, 1.5)
            descriptor = AudioUtil.generate_descriptors(audio_fast_3)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_15=%s WHERE id

            audio_slow_1 = librosa.effects.time_stretch(audio_slow_1, 0.9)
            descriptor = AudioUtil.generate_descriptors(audio_slow_1)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_16=%s WHERE id

            audio_slow_2 = librosa.effects.time_stretch(audio_slow_2, 0.8)
            descriptor = AudioUtil.generate_descriptors(audio_slow_2)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_17=%s WHERE id

            audio_slow_3 = librosa.effects.time_stretch(audio_slow_3, 0.5)
            descriptor = AudioUtil.generate_descriptors(audio_slow_3)
            byte_string = descriptor.tobytes()
            db_util.execute_insert_blob("UPDATE song SET des_18=%s WHERE id

        db_util.close()
        gc.collect()

    @staticmethod
    def generate_descriptors(audio_stream):
        # Generate STFT spectrogram
        stft_spectrogram = np.abs(librosa.stft(audio_stream, win_length=204
        librosa.display.specshow(librosa.amplitude_to_db(stft_spectrogram,
        plt.tight_layout()

        # Save spectrogram to File Buffer
        file_variable = io.BytesIO()
        plt.savefig(file_variable, bbox_inches='tight')
        plt.close()
        file_variable.seek(0)
        file_bytes = np.asarray(bytearray(file_variable.read()), dtype=np.u

        # Read File Buffer as OpenCV Image
        output_image = cv.imdecode(file_bytes, cv.IMREAD_COLOR)

        # Generate Descriptors
```

```
        sift = cv.xfeatures2d.SIFT_create()
        _, descriptors = sift.detectAndCompute(output_image, None)
        gc.collect()

        # Return Descriptors
        return descriptors
```

A.2.4 database-util.py

```
import mysql.connector
from mysql.connector import Error

class DatabaseUtil:
    def __init__(self):
        self.connection = None
        self.cursor = None

    def get_db_connection(self):
        if self.connection is None or not self.connection.is_connected():
            try:
                self.connection = mysql.connector.connect(host='localhost',
                                                          database='rms_usi
                                                          user='rms_db_user
                                                          password='pKeM_9#
                                                          ,use_pure=True)

            except Error as e:
                print("Database Error : ", e)

    def execute_query(self, query):
        if self.connection is None:
            self.get_db_connection()

        self.cursor = self.connection.cursor()
        print(query)
        self.cursor.execute(query)
        rows = self.cursor.fetchall()
        self.cursor.close()
        return rows

    def execute_insert(self, query):
        if self.connection is None:
            self.get_db_connection()

        self.cursor = self.connection.cursor()
        self.cursor.execute(query)
        self.connection.commit()
        rows = self.cursor.lastrowid
        self.cursor.close()
        return rows

    def execute_insert_blob(self, query, arg):
        if self.connection is None:
            self.get_db_connection()

        self.cursor = self.connection.cursor()
        self.cursor.execute(query, (arg,))
        self.connection.commit()
        rows = self.cursor.lastrowid
        self.cursor.close()
        return rows

    def execute_update_blob(self, query, arg):
        if self.connection is None:
            self.get_db_connection()

        self.cursor = self.connection.cursor()
        self.cursor.execute(query, (arg,))
        self.connection.commit()
        self.cursor.close()

    def execute_update(self, query):
```

```
        if self.connection is None:
            self.get_db_connection()

        self.cursor = self.connection.cursor()
        self.cursor.execute(query)
        self.connection.commit()
        self.cursor.close()

    @staticmethod
    def get_song_list():
        db_connection = None
        try:
            db_connection = mysql.connector.connect(host='localhost',
                                                    database='rmsdb',
                                                    user='rms_db_user',
                                                    password='pKeM_9#ryY')
        except Error as e:
            print("Database Error : ", e)
        if db_connection is not None:
            db_cursor = db_connection.cursor()
            db_cursor.execute("SELECT id, title FROM songs")
            rows = db_cursor.fetchall()
            db_cursor.close()
            db_connection.close()
            return rows
        return None

    def close(self):
        if self.connection is not None:
            self.connection.close()
        self.connection = None
```

A.2.5 find-count.py

```
from util.job_util import Job

if __name__ == '__main__':
    register_job = Job(Job.type["find_count"])
    print("Job Started. Job ID : " + str(register_job.job_id))
    register_job.start_job(None)
```

A.2.6 find-similar.py

```
from util.job_util import Job

if __name__ == '__main__':
    register_job = Job(Job.type["find_similar"])
    print("Job Started. Job ID : " + str(register_job.job_id))
    register_job.start_job(None)
```

A.2.7 job-util.py

```
from util.database_util import DatabaseUtil
from util.audio_util import AudioUtil


class Job:

    type = {"register": 1, "match": 2,"find_similar":3,"find_count":4}
    status = {"created": 1, "started": 2, "finished": 3, "errored": 4}

    def __init__(self, job_type):
        self.job_type = job_type
        db_util = DatabaseUtil()
        self.job_id = db_util.execute_insert("INSERT INTO job (type,status)
```

```
                                                + str(Job.status["created"]) +

    def start_job(self, filepath):
        if self.job_type == Job.type["register"]:
            AudioUtil.register_songs(self.job_id)
        if self.job_type == Job.type["match"]:
            AudioUtil.match_songs(self.job_id)
        if self.job_type == Job.type["find_similar"]:
            AudioUtil.find_similar_songs(self.job_id)
        if self.job_type == Job.type["find_count"]:
            AudioUtil.find_count(self.job_id)
```