Sentiment Analysis from Speech using Feature Extraction

by

AVINANDAN MUKHERJEE (Roll: 544)
SHAHMEER MONDAL (Roll: 556)
SRIJAN SEN (Roll: 523)

under the guidance of

DR. SONALI SEN

Submitted to the Department of Computer Science in partial fulfilment of the requirements for the Degree of B.Sc. Computer Science. We affirm that we have identified all our sources and that no part of our dissertation paper uses unacknowledged materials.

St. Xavier's College (Autonomous), Kolkata
30, Mother Teresa Sarani, Kolkata - 700016.
Department of Computer Science
www.sxccal.edu | csc@sxccal.edu | 2255 1271

CERTIFICATE OF AUTHENTICATED WORK

This is to certify that the project report entitled **"Sentiment Analysis from Speech using Feature Extraction"** submitted to the Department of Computer Science, ST. XAVIER'S COLLEGE [AUTONOMOUS], KOLKATA, in partial fulfilment of the requirement for the award of the degree of Bachelor of Science (B.Sc.) is an original work carried out by:

Name	**Roll Number**	**Registration Number**
Avinandan Mukherjee	544	A01-1112-0932-21
Shahmeer Mondal	556	A01-1142-0942-21
Srijan Sen	523	A01-1112-0917-21

under my guidance. The matter embodied in this project is authentic and is genuine work done by the student and has not been submitted whether to this College or to any other Institute for the fulfilment of the requirement of any course of study.

....

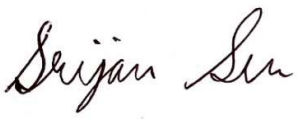
Signature of the Supervisor Signature of the Head of the Department

.....

Name of the Supervisor

Date: ..... Date: .....

Name and Signature of the **Project Team** members:

Serial No.	**Name**	**Signature**
1	Avinandan Mukherjee	
2	Shahmeer Mondal	
3	Srijan Sen	

IMPLEMENTATION RESPONSIBILITIES

Roll Number	Name	Responsibility	Section of Report
544	Avinandan Mukherjee	Conceptual Models	4.1
		Model Training and Validation	5.1.8 to 5.1.10
		Unit & Integration Testing	5.3 & 5.4
		Analysis of Dataset	6.1
		Comparative Study of used Models	6.2
556	Shahmeer Mondal	Procedural Design	4.2
		Feature Extraction	5.1.5 to 5.1.7
		Model Loading and Unknown Sample Testing	5.1.11 to 5.1.12
		Comparative Study of used Models	6.2
		Classification Report	6.3
523	Srijan Sen	Dataset Handling	5.1.1
		Data Pre-processing	5.1.2 to 5.1.4
		Unit Testing	5.3
		Integration Testing	5.4
		Analysis of Dataset	6.1

ACKNOWLEDGEMENT

We would like to take this occasion to express our heartfelt gratitude to everyone who has helped and guided us throughout the development and completion of this project.

We express our gratitude to the Head of the Department of Computer Science for providing us with the chance to engage in this project, which has enabled us to delve into the captivating world of machine learning and gain insights into the technologies utilised herein.

We are thankful to our project mentor, Dr. Sonali Sen, for expertly guiding us through the different phases of this project, and for elucidating major concepts with clarity.

We extend our sincere appreciation to all the teaching and non-teaching staff of the Department of Computer Science for their unwavering support and assistance, without which the project would not have been completed.

Lastly, we extend our immense gratitude to our parents for enabling us to become part of this esteemed institution.

ABSTRACT

In recent years, voice sentiment analysis has emerged as a pivotal area of research with diverse applications spanning from customer service to mental health diagnostics. This project delves into the development and application of a novel sentiment analysis system tailored specifically for analysing speech patterns and inflections to discern underlying emotional states. Through the integration of machine learning techniques and signal processing algorithms, our framework aims to accurately classify spoken content into seven distinct sentiment categories, thereby enabling a deeper understanding of human emotions conveyed through speech.

The project uses a database of audio recordings encompassing a wide spectrum of emotions and variations in accents. These recordings serve as the foundation for training and fine-tuning our sentiment analysis model. Additionally, feature extraction techniques are employed to extract relevant acoustic features, which play a crucial role in capturing subtle nuances in vocal expressions. All this information is used by the model to predict the emotion encoded in a given audio file.

TABLE OF CONTENTS

CONTENTS	PAGE
CHAPTER 1: INTRODUCTION	**10**
1.1 Background	10
1.2 Objective	10
1.3 Purpose	10
1.4 Scope	11
1.5 Applicability	11
1.6 Achievements	12
CHAPTER 2: SURVEY OF TECHNOLOGIES	**13**
2.1 Mel Frequency Cepstral Coefficients	13
2.2 Principal Component Analysis	16
2.3 Support Vector Machine	18
2.4 K-Nearest Neighbour Algorithm	19
2.5 Neural Networks	21
2.5.1 Feedforward Neural Networks	22
2.5.2 Convolutional Neural Networks	22
2.5.3 Recurrent Neural Networks	22
2.5.4 Gated Recurrent Unit	23
2.5.5 Long Short-Term Memory	24
2.5.6 Adam’s Optimizer	25

CHAPTER 3: REQUIREMENT ANALYSIS **27**

3.1 Problem Definition 27

3.2 Software Requirements 27

3.3 Hardware Requirements 29

CHAPTER 4: SYSTEM DESIGN **30**

4.1 Conceptual Models 30

4.2 Procedural Design 33

4.3 About the Datasets 35

4.3.1 SAVEE Dataset 35

4.3.2 RAVDESS Dataset 36

4.3.3 TESS Dataset 37

4.3.4 CREMA-D Dataset 38

CHAPTER 5: CODING AND TESTING **39**

5.1 Coding Details 39

5.1.1 Importing Libraries 39

5.1.2 Dataset Handling 40

5.1.3 Data Pre-Processing and Dataset Exploring 41

5.1.4 Data-frames Handling 43

5.1.5 Feature Extraction 45

5.1.6 Splitting of Training and Testing Datasets 46

5.1.7 Z-score Normalisation 47

5.1.8 KNN Model Training and Validation 47

5.1.9 SVM Model Training and Validation 51

5.1.10 LSTM Model Training and Validation 52

5.1.11 LSTM Loading 56

5.1.12 Testing an Unknown Sample 57

5.2 Complete Source Code 59

5.3 Unit Testing 70

5.4 Unit Testing using 7 classes 80

5.5 Integration Testing 82

CHAPTER 6: RESULTS AND DISCUSSIONS 83

6.1 Analysis of the Dataset 83

6.2 Comparative study of the used Models 85

6.3 Classification Report of LSTM 89

6.4 Final Results 95

CHAPTER 7 96

7.1 Limitations of the Project 96

7.2 Future Scope 96

REFERENCES and CITATIONS 99

TABLE OF FIGURES, TABLES & ALGORITHMS

CONTENTS **PAGE**

FIGURES

Fig 2.1.1: Feature Extraction of a sample audio using MFCC 13

Fig 2.1.2: MFCC Flowchart 15

Fig 2.2.1: A PCA Scatter Plot for a sample audio file 16

Fig 2.3.1: A SVM two-dimensional feature space 19

Fig 2.4.1: KNN Algorithm Logic 20

Fig 2.5.1: A simple Neural Network 21

Fig 2.5.2: Basic RNN Architecture (Unfolded) 22

Fig 2.5.3: A basic LSTM Unit 24

Fig 2.5.4: A Plot showing Adam's Optimizer function 26

Fig 4.1.1: A flowchart showing the Dataset Handling 30

Fig 4.1.2: A flowchart showing the Feature Extraction 31

Fig 4.1.3: Model Creation, Training & Testing 32

Fig 4.1.4: A flowchart showing the Full Working 33

Fig 4.3.1: Emotion Distribution in SAVEE Dataset 36

Fig 4.3.2: Emotion Distribution in RAVDESS Dataset 37

Fig 4.3.3: Emotion Distribution in TESS Dataset 37

Fig 4.3.4: Emotion Distribution in CREMA-D Dataset 38

Fig 4.3.5: Emotion Distribution in the combined Dataset 38

Fig 6.1.1: Emotion distribution in the Combined Dataset 84

ALGORITHMS

Algo 4.2.1: Feature Extraction using MFCC 34

Algo 4.2.2: LSTM Model Creation 34

Algo 4.2.3: LSTM Model Training and Testing 35

TABLES

Table 6.1.1: Audio Dataset Samples 83

CHAPTER 1: INTRODUCTION

1.1 Background

Theodoros Giannakopoulos [1] in 2015, introduced pyAudioAnalysis, an open-source Python library for analysing audio content. It offered various features like feature extraction, signal classification, and content visualisation. The library has been used in diverse applications such as smart-home automation, speech emotion recognition, and health monitoring, leading to continual improvements based on user feedback.

Zhiyun Lu, Liangliang Cao, Yu Zhang, Chung-Cheng Chiu, James Fan [2] in 2020, wrote a paper which proposed leveraging pre-trained features from end-to-end automatic speech recognition (ASR) models for speech sentiment analysis. The integration of acoustic and textual information from speech enabled promising results. A sentiment classifier based on recurrent neural networks (RNN) with self-attention was utilised, facilitating model interpretation through attention weights visualisation. Evaluation was conducted on the IEMOCAP dataset and a new SWBD-sentiment dataset. The proposed approach achieved significant improvements in accuracy, enhancing state-of-the-art performance on IEMOCAP from 66.6% to 71.7%, and attaining an accuracy of 70.10% on SWBD-sentiment with over 49,500 utterances.

Tapesh Kumar, Mehul Mahrishi & Sarfaraz Nawaz [3] in 2022 wrote a paper to start meaningful research into speech sentiment analysis without the help of text sentiment analysis. The paper proposed an investigation into speech sentiment analysis, incorporating speaker recognition to evaluate emotional expressions of speakers across various transcripts. Understanding the sentiment conveyed through speech did hold substantial importance in numerous sectors, including customer service, healthcare, and education, where effective communication and emotional understanding play pivotal roles in enhancing user experience, patient care, and educational outcomes.

1.2 Objective

The objective of this project is to develop an accurate and robust speech sentiment analysing system using machine learning techniques. Through the analysis of audio data, the system aims to classify human emotions into seven distinct categories, facilitating applications in human-computer interaction, mental health monitoring, and affective computing.

1.3 Purpose

A project on speech sentiment analysis is crucial due to its profound impact on various aspects of human-computer interaction and societal well-being. Understanding emotions

conveyed through speech is essential for developing empathetic and responsive AI systems, such as virtual assistants and customer service chatbots. By accurately discerning emotional states from voice data, these systems can tailor their responses to better meet user needs, enhancing user satisfaction and engagement.

Moreover, speech sentiment analysis holds significant implications for mental health diagnostics and support. By analysing vocal cues, such as tone of voice and intonation, researchers and clinicians can gain insights into an individual's emotional state, aiding in the early detection of mood disorders or psychological distress. This proactive approach to mental health monitoring can lead to timely interventions and improved patient outcomes.

Additionally, in fields such as market research and product development, speech sentiment analysis enables companies to gauge consumer opinions and preferences more effectively. By analysing customer feedback from call centre interactions or social media platforms, businesses can identify areas for improvement, refine their products or services, and tailor marketing strategies to better resonate with their target audience.

In essence, a project on speech sentiment analysis is not only important but also necessary for advancing human-computer interaction, mental health care, and market intelligence. By leveraging the power of machine learning and signal processing techniques, researchers and practitioners can unlock new opportunities for enhancing communication, empathy, and societal well-being.

1.4 Scope

The project endeavours to analyse audio files for emotion detection, assigning them to one of seven predetermined emotion classes. Leveraging feature extraction techniques, it extracts features from the audio data to discern underlying emotional states. The system provides a valuable tool for understanding emotional expressions conveyed through speech. However, its scope extends beyond mere classification; it offers insights applicable to diverse domains such as customer service, mental health assessment, and market research. The project faces limitations inherent to speech analysis, including variability in speech patterns and environmental factors, which may affect the precision of emotion classification.

Moreover, assumptions underpinning the project's methodology include the consistency of emotional expressions within the database and the quality of audio recordings. These assumptions play a crucial role in shaping the system's performance and effectiveness.

1.5 Applicability

The applications of this project span a wide range of domains, leveraging automated emotion detection from audio files to enhance various aspects of human-computer interaction

and societal well-being. In customer service settings, the ability to accurately discern emotions from spoken interactions enables businesses to provide more personalised and empathetic support to customers. By analysing the emotional tone of customer calls, companies can identify and prioritise urgent issues, leading to improved customer satisfaction and loyalty. Additionally, sentiment analysis in call centre environments can help optimise staffing levels and training programs, ensuring that customer inquiries are handled efficiently and effectively.

In the realm of mental health care, the project holds significant promise for early detection and intervention in mood disorders and psychological distress. By analysing vocal cues indicative of emotional states, clinicians can gain insights into patients' mental well-being remotely, facilitating timely interventions and personalised treatment plans. Moreover, the project's application in telemedicine and remote patient monitoring enables access to mental health support in underserved communities and during times of crisis, bridging gaps in access to care and improving overall mental health outcomes.

Beyond customer service and mental health, the project's applications extend to market research and product development. By analysing consumer sentiment expressed in audio feedback, companies can gain valuable insights into consumer preferences, trends, and opinions. This enables them to tailor marketing strategies, refine product offerings, and identify areas for innovation, ultimately driving business growth and competitiveness in the marketplace.

Furthermore, in educational settings, the project can be utilised to assess student engagement and emotional well-being during remote learning sessions. By analysing audio recordings of classroom interactions, educators can identify students who may require additional support or intervention, fostering a more inclusive and supportive learning environment.

Overall, the applications of this project are diverse and far-reaching, spanning industries such as customer service, mental health care, market research, and education. By harnessing the power of automated emotion detection from audio files, it has the potential to revolutionise how we understand, interact with, and support one another in various facets of life.

1.6 Achievements

In this project, we aimed to analyse the emotional content of audio files through feature extraction using MFCC. After extracting features from the audio inputs, we trained and validated multiple machine learning models. Ultimately, the LSTM model emerged as the most effective, achieving an accuracy ranging from 58 to 60%. This approach allowed us to decode emotional nuances within the audio data, offering insights into sentiment and expression.

CHAPTER 2: SURVEY OF TECHNOLOGIES

2.1 Mel Frequency Cepstral Coefficients

Mel Frequency Cepstral Coefficients (MFCC), a key feature extraction method in speech and audio processing, plays a vital role in tasks like speech recognition and music analysis. Essentially, MFCCs encapsulate the spectral characteristics of sound in a format well-suited for machine learning applications. To put it simply, they are coefficients that encapsulate the power spectrum shape of a sound signal.

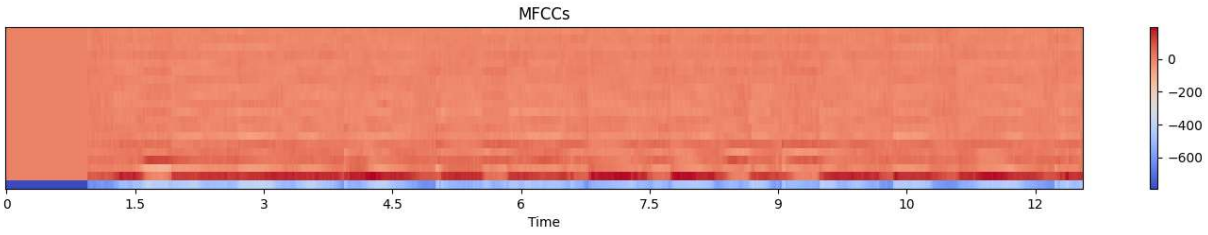


Fig 2.1.1: Feature Extraction of a sample audio using MFCC

The audio signal is pre-emphasized to amplify higher frequencies, which improves the signal-to-noise ratio. This is done by applying a first-order high-pass filter to the signal:

$$y(t) = x(t) - \alpha \cdot x(t-1)$$

where,

x(t) is the input signal,
y(t) is the pre-emphasized signal, and
α is the pre-emphasis coefficient (typically around 0.97).

The pre-emphasized signal is divided into short frames of typically 20-40 milliseconds, with overlap between adjacent frames. Each frame is windowed using a window function such as the Hamming window to reduce spectral leakage.

$$x_{\text{pre-emphasized}}[n] = x[n] - \text{pre-emphasis\_factor} \times x[n-1]$$

where, x[n] is the input audio signal.

The windowing operation is represented as:

$$x_{\text{windowed}}[n] = x_{\text{frame}}[n] \times w[n]$$

where, w[n] is the windowing function.

Discrete Fourier Transform (DFT) is applied to each frame to convert the signal from the time domain to the frequency domain. This results in a power spectrum representation of the signal.

$$X(k) = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}$$

where,

X(k) is the k-th frequency component,
x[n] is the windowed signal,
N is the number of samples, and
K ranges from 0 to N - 1.

The power spectrum is then passed through a bank of Mel filters, which are triangular filters spaced evenly on the Mel frequency scale. The Mel scale approximates the human auditory system's perception of frequency. It is used to map the actual frequency to the frequency that human beings perceive. The output of each filter is the sum of the power spectrum weighted by the filter's shape.

$$mel(f) = 1127 \ln(1 + \frac{f}{700})$$

The logarithm of the Mel-filtered spectrum is computed to mimic the logarithmic perception of loudness by the human auditory system. This is done to linearize the frequency axis.

$$X_{\log}(m) = \log(X(m))$$

The resulting Mel-frequency log spectra are passed through a Discrete Cosine Transform (DCT) to decorrelate the features and compact most of the energy into a small number of coefficients. Typically, only the lower-frequency DCT coefficients are retained. Mathematically, the DCT is defined as:

$$C_{\text{coeff}}(i) = \sum_{n=0}^{N-1} X_{\log}(n) \cdot \cos\left(\frac{\pi}{N}(n + 0.5)i\right)$$

where,

$C_{coeff}(i)$ is the i-th MFCC coefficient,
$X_{log}(n)$ is the log filterbank energy,
N is the number of filters, and
i ranges from 0 to M - 1.

The complete flowchart of MFCC is given in the next page:

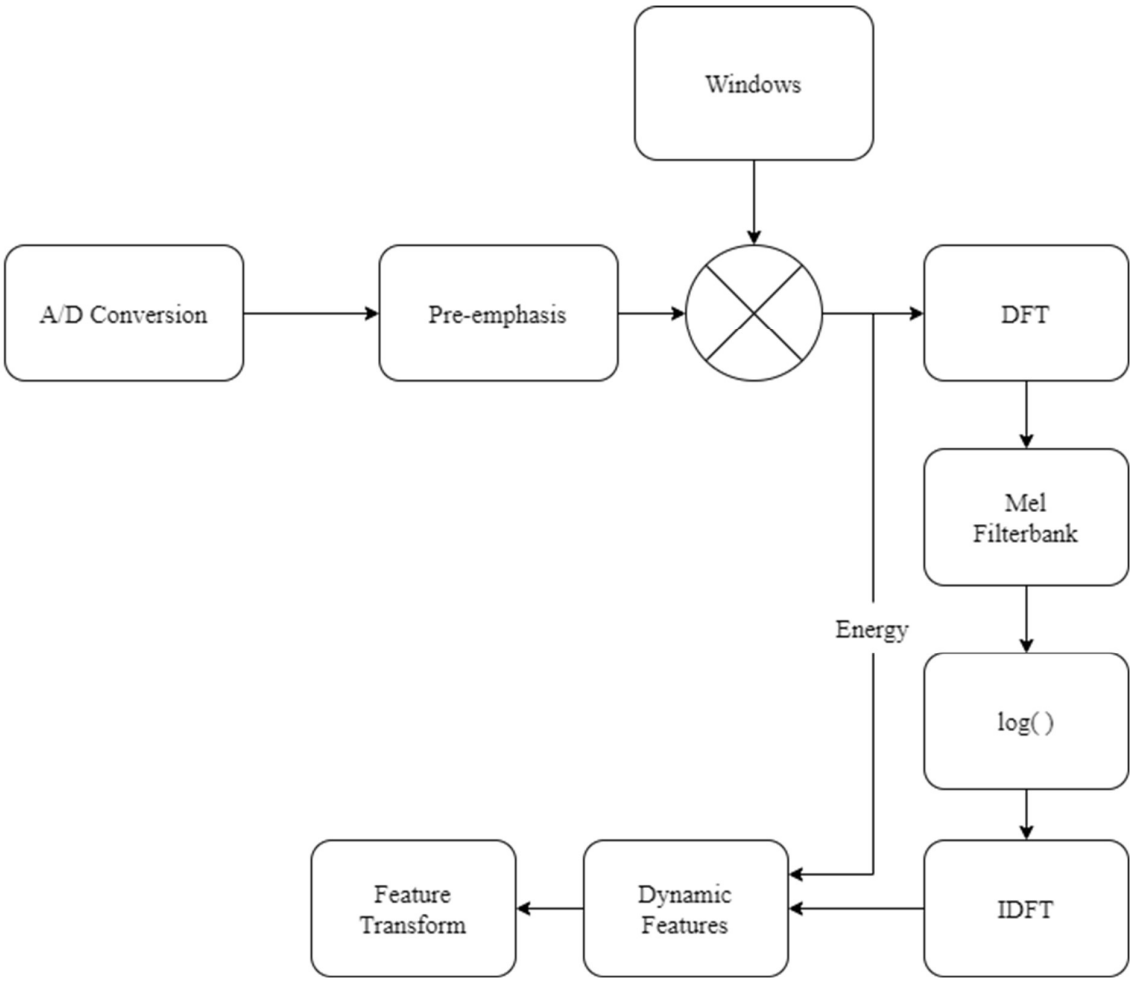


Fig 2.1.2: MFCC Flowchart

Mathematically, the MFCC computation can be summarised as follows:

$$c_m = \sum_{k=1}^{N} \log \left(\sum_{n=1}^{N} x(n) \cdot h_m(n-k) \right) \cdot \cos \left[\frac{\pi m(k-0.5)}{N} \right]$$

where,

N is the number of samples in each frame,
$h_m(n)$ is the Mel filterbank,
x(n) is the pre-emphasized signal,
M is the MFCC index.

MFCCs are valuable because they highlight aspects of the audio signal crucial for human speech perception, making them ideal for applications such as speaker recognition, emotion detection, and speech-to-text conversion.

2.2 Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique used to transform high-dimensional data into a lower-dimensional space while preserving the most important information. Mathematically, PCA involves finding the eigenvectors and eigenvalues of the covariance matrix of the data.

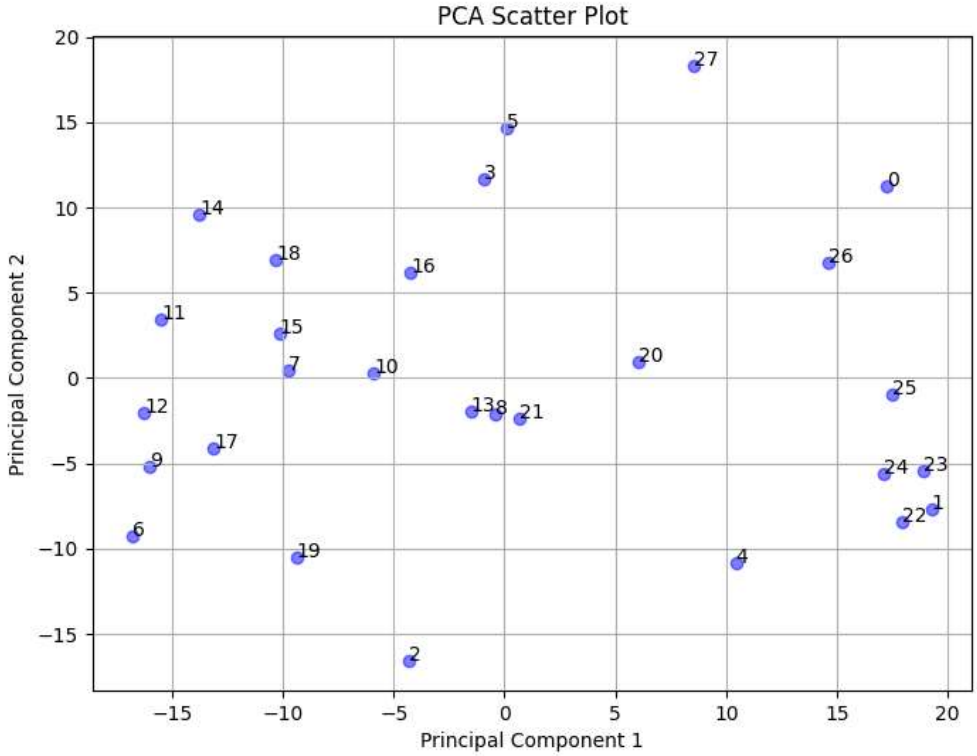


Fig 2.2.1: A PCA Scatter Plot for a sample audio file

PCA is a mathematical technique that reorients data in a new coordinate system, emphasising the directions where the data varies the most. This transformation ensures that the first coordinate captures the highest variance, followed by subsequent coordinates in decreasing order of variance.

Let *X* be an *n x m* matrix representing *n* observation of *m* features. The first step in PCA is to centre the data by subtracting the mean of each feature from the corresponding column in *X*, resulting in the centred data matrix $\bar{X}$.

Next, PCA computes the covariance matrix Σ of *X*, given by:

$$\Sigma = \frac{1}{n-1}\bar{X}^T\bar{X}$$

The covariance matrix captures the pairwise relationships between the features. PCA then calculates the eigenvectors and eigenvalues of Σ. The eigenvectors represent the principal components, and the eigenvalues indicate the amount of variance explained by each component.

Let v_1, v_2, ..., v_m denote the eigenvectors of Σ, arranged in descending order of their corresponding eigenvalues λ_1, λ_2, ..., λ_m. The principal components PC_1, PC_2, PC_3, ..., PC_m are obtained by projecting the centred data onto the eigenvectors:

$$PC_i = \bar{X} v_i \quad \text{for} \quad i = 1, 2, ..., m$$

After computing the eigenvectors $v_1, v_2, ..., v_m$ and their corresponding eigenvalues $\lambda_1, \lambda_2, ..., \lambda_m$, PCA sorts them in descending order based on the eigenvalues. This sorting reflects the importance of each principal component in capturing the variance within the data.

To select the top k principal components, PCA considers the cumulative explained variance ratio, denoted by VAR(k), which represents the proportion of total variance explained by the first k principal components. It is computed as follows:

$$\text{VAR}(k) = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{m} \lambda_i}$$

Once the desired number of principal components k is determined, PCA projects the centred data matrix X' onto the selected principal components to obtain the reduced-dimensional representation. Let Vk denote the matrix containing the first k eigenvectors (principal components) stacked as columns. The projection of X' onto V_k yields the reduced-dimensional data matrix $X_{reduced}$:

$$X_{\text{reduced}} = \bar{X} V_k$$

This reduced-dimensional representation $X_{reduced}$ can be used for data reconstruction by multiplying it with the transpose of V_k:

$$\hat{X} = X_{\text{reduced}} V_k^T$$

where X^ represents the reconstructed data matrix. Additionally, $X_{reduced}$ can serve as input for further analysis tasks such as clustering or classification in the reduced-dimensional space.

In summary, PCA facilitates dimensionality reduction by selecting the top k principal components that capture the most variance in the data. These components define a new coordinate system, allowing for efficient representation, data reconstruction, and subsequent analysis while preserving the essential information contained in the original data.

2.3 Support Vector Machine

Support Vector Machine (SVM) is a supervised learning algorithm used for classification and regression tasks. It's particularly effective in scenarios where there's a clear separation between classes in the feature space. SVM works by finding the optimal hyperplane that separates data points of different classes with the maximum margin. This hyperplane is defined by a set of support vectors, which are the data points closest to the decision boundary.

Mathematically, given a training dataset:

$$\{(x_i, y_i)\}_{i=1}^{n}$$

where x_i represents the input features and y_i represents the class labels (either -1 or 1 for binary classification), the objective of SVM is to find the optimal hyperplane:

$$w^T x + b = 0$$

that separates the data points:

$$y_i(w^T x_i + b) \geq 1 \quad \text{for all} \quad i = 1, 2, ..., n$$

Here, w represents the weights vector and b represents the bias term. SVM aims to maximise the margin, which is the distance between the hyperplane and the support vectors, while minimising classification error.

In cases where the data is not linearly separable, SVM utilises the kernel trick to transform the input features into a higher-dimensional space where they become linearly separable. This is achieved by replacing the dot product $T(x_i)x_j$ with a given kernel function $K(x_i, x_j)$, such as the polynomial kernel or Gaussian (RBF) kernel. The kernel trick allows SVM to handle nonlinear decision boundaries effectively.

The decision boundary of SVM is determined by the support vectors, which are the data points that lie closest to the hyperplane. These support vectors are crucial for defining the optimal separating hyperplane and are used to classify new data points.

SVMs have several advantages, including their effectiveness in high-dimensional spaces, their ability to handle non-linear decision boundaries through kernel functions, and their resistance to overfitting. However, SVMs can be computationally expensive, especially when dealing with large datasets.

To visualise how SVM works, consider a two-dimensional feature space with two classes of data points. The goal of SVM is to find the line (in two dimensions) or hyperplane (in higher dimensions) that separates these classes with the maximum margin. The support

vectors, which are the data points closest to the decision boundary, define this margin. By maximising the margin, SVM ensures a robust decision boundary that generalises well to unseen data.

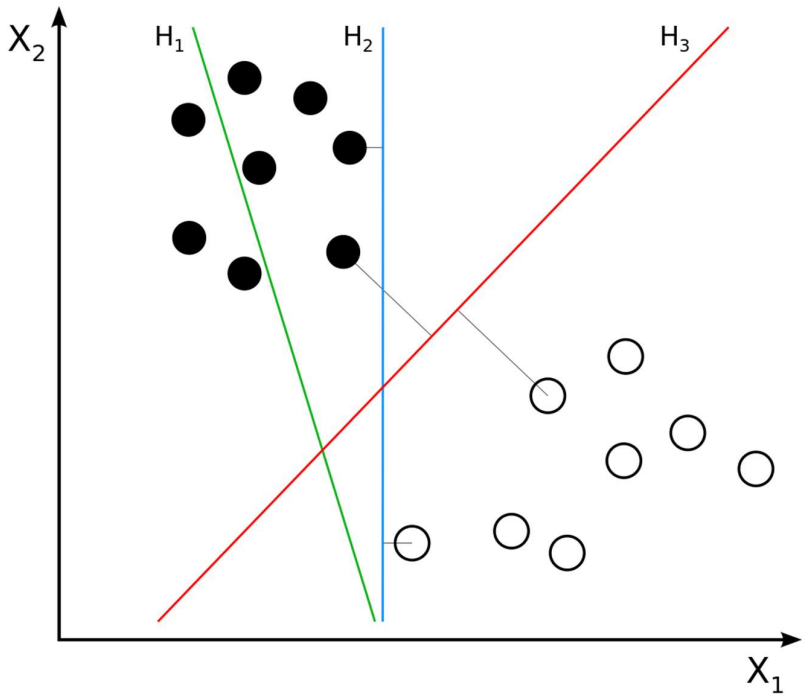


Fig 2.3.1: A SVM two dimensional feature space with two classes of data points and different hyperplanes separating the two classes with a certain margin. Hyperplane H1 does not separate the classes. Hyperplane H2 does, but only with a small margin. Hyperplane H3 separates them with the maximal margin. [7-i]

In summary, SVM is a versatile and powerful machine learning algorithm used for classification and regression tasks. It works by finding the optimal hyperplane that separates data points of different classes with the maximum margin, using support vectors to define this boundary. SVMs are widely used in various applications such as text classification, image recognition, and bioinformatics.

2.4 K-Nearest Neighbour Algorithm

The K-Nearest Neighbors (KNN) algorithm is a simple and versatile machine learning algorithm used for classification and regression tasks. It operates based on the principle that similar instances are likely to belong to the same class or have similar target values. KNN stores the entire training dataset, consisting of labelled instances with their corresponding class labels or target values.

For a given unlabeled data point x(new), the algorithm calculates the distance between this point and all the data points in the training dataset. The most common distance metric used is the Euclidean distance, given by:

$$\text{Distance}(x_{\text{new}}, x_i) = \sqrt{\sum_{j=1}^{d} (x_{\text{new},j} - x_{i,j})^2}$$

where d is the number of features.

It then identifies the *K* nearest neighbours of the unlabeled data point based on the calculated distances. *"K"* is a hyperparameter specified by the user. For classification tasks, KNN assigns the majority class among the *K* nearest neighbours to the unlabeled data point. In other words, the predicted class label of the test instance is determined by the most frequently occurring class among its *K* nearest neighbours. For regression tasks, KNN calculates the average (or weighted average) of the target values of the *K* nearest neighbours and assigns it as the predicted value for the unlabeled data point.

The choice of the hyperparameter *K* is crucial in KNN, as it significantly affects the model's performance. A smaller value of *K* may lead to overfitting, while a larger value may result in underfitting. The value of *K* can be determined through hyperparameter tuning techniques such as cross-validation.

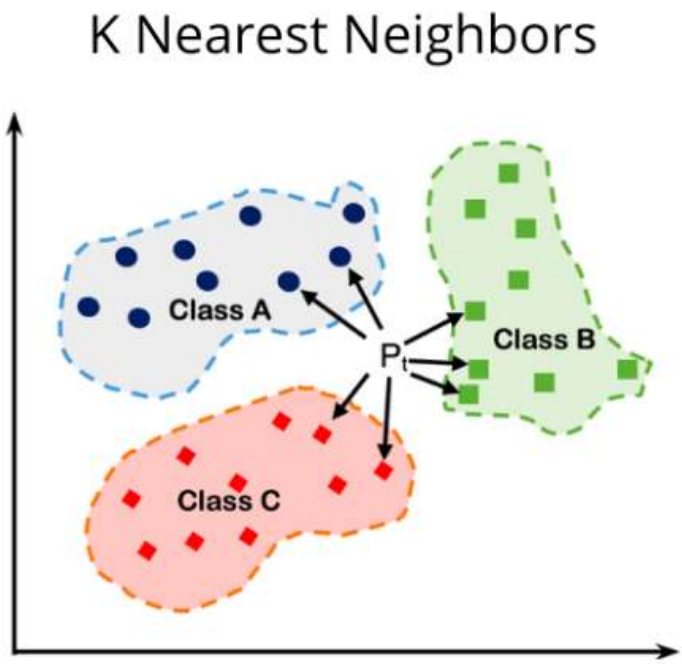


Fig 2.4.1: KNN Algorithm Logic [8]

Indeed, the computational complexity of the KNN algorithm during the prediction phase can be a significant drawback, particularly when dealing with large datasets. Since KNN requires storing the entire training dataset in memory, making predictions for new instances

involves calculating distances from the new instance to all data points in the training set. This process can become computationally expensive as the size of the dataset grows.

Furthermore, the time complexity of KNN for making predictions increases linearly with the size of the training dataset. For each new instance, the algorithm needs to compute distances to all existing data points, sort them to identify the nearest neighbours, and then make a prediction based on these neighbours. This process can lead to longer prediction times, especially when dealing with high-dimensional data or datasets with a large number of instances.

So, KNN struggles with scalability when dealing with extremely large datasets or high-dimensional data. In such cases, alternative machine learning algorithms with better scalability properties may be more suitable.

2.5 Neural Networks

Neural networks are a class of machine learning algorithms inspired by the structure and function of the human brain. They consist of interconnected nodes, called neurons, organised in layers. Each neuron receives input signals, processes them using an activation function, and produces an output signal. Neural networks are trained using labelled data to learn patterns and relationships within the data.

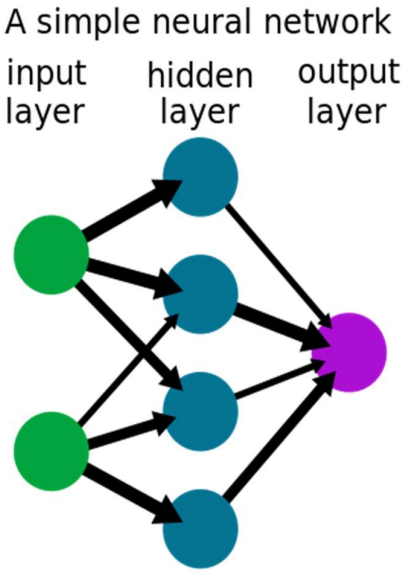


Fig 2.5.1: A simple Neural Network [7 ii]

There are several types of neural networks namely Feedforward Neural Networks (FNN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) and Long Short-Term Memory Networks (LSTM).

2.5.1 Feedforward Neural Networks

Feedforward Neural Networks (FNN), also known as artificial neural networks, are the simplest type of neural network architecture. In FNNs, information flows in one direction, from input nodes through hidden layers to output nodes, without any loops or cycles. Each layer consists of neurons that process input data using weighted connections and activation functions, producing output signals. FNNs are commonly used for supervised learning tasks such as classification and regression, where they learn to map input data to output labels by adjusting the weights and biases during training using algorithms like gradient descent.

2.5.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are specialised neural network architectures designed for processing grid-like data, such as images. CNNs consist of convolutional layers that apply filters to input data, capturing spatial hierarchies of features. These layers are typically followed by pooling layers, which reduce the spatial dimensions of the features while retaining important information. CNNs are well-suited for tasks like image recognition, object detection, and image segmentation, where they can automatically learn and extract hierarchical representations of features from raw image data, leading to highly accurate predictions.

2.5.3 Recurrent Neural Networks

Basic neural networks struggle with variable-length inputs and outputs, lacking the ability to share learned features across positions in sequences. Recurrent Neural Networks (RNNs) address these issues by processing sequences through time. They use hidden states to retain information about previous steps, enabling memory and contextual understanding.

RNNs predict output based on input from both the current and previous time steps, enabling tasks like next-word prediction. They process data sequentially, incorporating information from previous steps to enhance understanding. Bi-directional RNNs scan data in both directions to capture context from both past and future words.

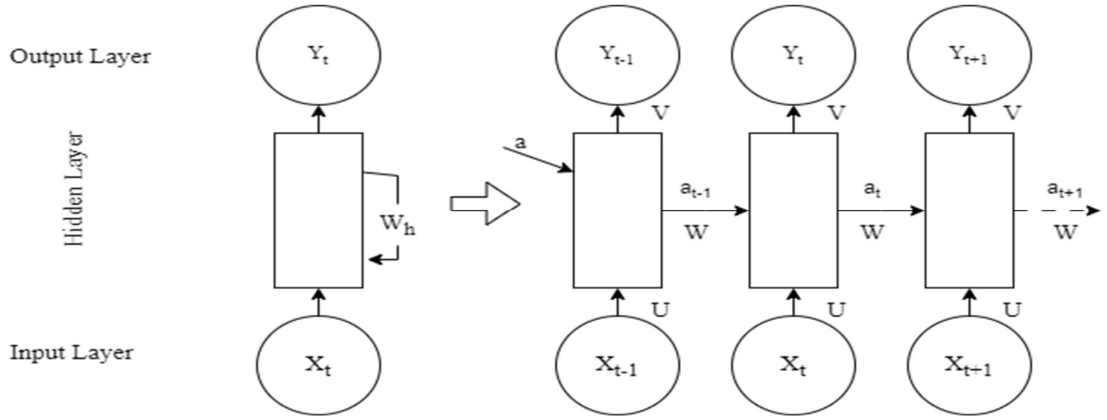


Fig 2.5.2: Basic RNN Architecture (Unfolded)

At each time step t, given an input sequence x_t (a vector representing the input at time t) and a hidden state a_{t-1} or h_{t-1} (a vector representing the information from the previous time step), the RNN computes the hidden state a_t or h_t and the output y_t as follows:

$$h^{(t)} = \sigma(W_{hx}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$
$$y^{(t)} = \text{softmax}(W_{yh}h^{(t)} + b_y)$$

where,

W_{hx} is the weight matrix connecting the input $x(t)$ to the hidden state $h(t)$.
W_{hh} is the weight matrix connecting the hidden state h^{t-1} to the current hidden state h^t.
b_h is the bias vector for the hidden state.
σ is the activation function (commonly the sigmoid or tanh function).
W_{yh} is the weight matrix connecting the hidden state h^t to the output y^t.
b_y is the bias vector for the output.
softmax is the softmax function used to compute probabilities for the output.

This recursive calculation allows the RNN to capture temporal dependencies and process sequential data effectively, making it suitable for tasks such as time series prediction, natural language processing, and speech recognition.

RNNs excel at capturing short-term dependencies but struggle with long-term ones. They are adept at incorporating local influences, where outputs are influenced by nearby inputs.

2.5.4 Gated Recurrent Unit

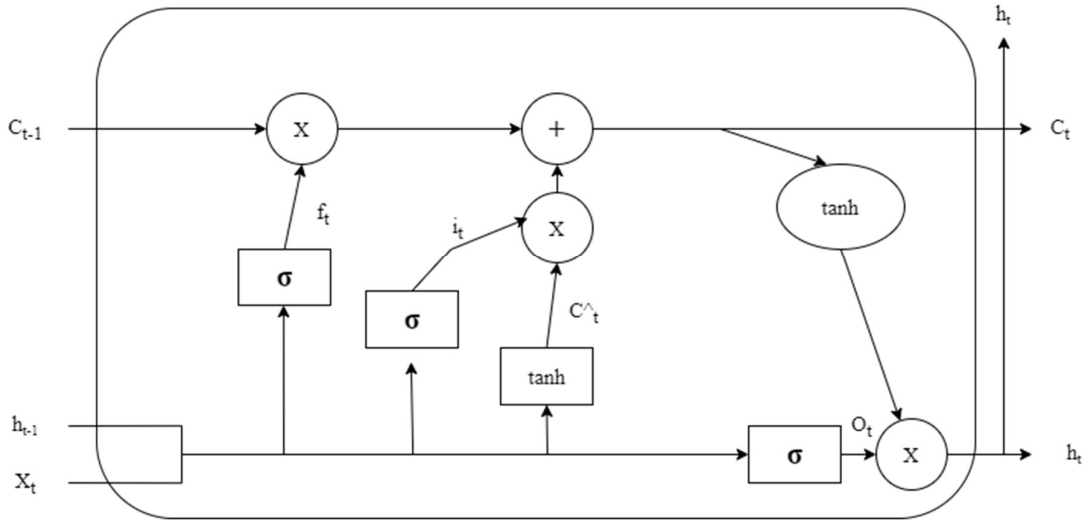
Gated Recurrent Units (GRUs) are a type of recurrent neural network (RNN) architecture that addresses the vanishing gradient problem and allows for better capture of long-term dependencies in sequential data. GRUs utilise gating mechanisms to regulate the flow of information within the network, consisting of reset and update gates. The reset gate controls how much past information to forget, while the update gate determines how much new information to incorporate.

In speech emotion recognition, GRUs are beneficial for capturing temporal dependencies and contextual information in speech signals. By effectively modelling the temporal dynamics of speech data, GRUs can extract relevant features and patterns associated with different emotional states expressed in speech. Their ability to retain and utilise long-term contextual information makes them well-suited for tasks requiring memory retention and sequential data processing, enhancing the accuracy and robustness of speech emotion recognition systems.

2.5.5 Long Short Term Memory

LSTM (Long Short-Term Memory) networks are an improvement over GRUs (Gated Recurrent Units) in the realm of recurrent neural networks (RNNs). While both GRUs and LSTMs address the vanishing gradient problem and enable better capture of long-term dependencies in sequential data, LSTMs offer additional mechanisms for memory retention and control, leading to more effective modelling of complex sequences.

Compared to GRUs, LSTMs have an additional cell state that runs through the entire sequence, allowing for better preservation of long-term information. LSTMs achieve this through three gates: the input gate, forget gate, and output gate. The input gate controls the flow of new information into the cell state, while the forget gate regulates which information to discard from the cell state. Finally, the output gate controls the information flow from the cell state to the output.



2.5.3: A basic LSTM Unit

At each time step t, an LSTM cell receives input x_t and the previous hidden state h_{t-1}. It computes three gates: the forget gate f_t, input gate i_t, and output gate o_t, along with a candidate cell state $\sim C_t$.

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
\tilde{C}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)
\end{aligned}
$$

The forget gate determines what information to discard from the cell state, while the input gate controls what information to update. The candidate cell state $\sim C_t$ is a new candidate

value to add to the cell state C_{t-1}, computed using the hyperbolic tangent function. Finally, the cell state C_t is updated as follows:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

The output gate o_t determines what part of the cell state should be output as the hidden state h_t, which carries information to the next time step and to the final prediction. LSTMs excel at capturing long-range dependencies and are widely used in tasks such as speech recognition, language modelling, and time series prediction.

In conclusion, LSTM (Long Short-Term Memory) networks outperform traditional RNNs and GRUs (Gated Recurrent Units) in speech emotion recognition tasks. With its ability to capture long-term dependencies and effectively retain contextual information, LSTM excels in modelling the temporal dynamics of speech data, leading to more accurate and robust emotion recognition systems.

2.5.6 Adam's Optimizer

Adam, short for Adaptive Moment Estimation, is an optimization algorithm commonly used in training deep learning models. It combines the concepts of momentum and adaptive learning rates to efficiently navigate the parameter space and converge towards the optimal solution.

At its core, Adam maintains two moving averages: the first moment (mean) of the gradients and the second moment (uncentered variance) of the gradients. These moving averages are calculated using exponentially decaying averages of past gradients, with momentum-like behaviour that accelerates the optimization process.

During training, Adam adapts the learning rate for each parameter individually based on the magnitudes of the gradients and the moving averages of past gradients. This adaptive learning rate mechanism allows Adam to handle varying gradients across different parameters, leading to faster convergence and improved performance.

Additionally, Adam incorporates bias correction to mitigate the effect of the initial estimates of the moving averages, particularly during the early stages of training when the estimates may be inaccurate.

The algorithm's hyperparameters include the learning rate, β1 (decay rate for the first moment), β2 (decay rate for the second moment), and ε (a small constant to prevent division by zero). These hyperparameters control the behaviour of the algorithm and can influence its performance on different tasks and datasets.

Overall, Adam's combination of momentum and adaptive learning rates makes it well-suited for training deep neural networks, offering fast convergence, robustness to noisy

gradients, and ease of use due to its automatic adaptation of learning rates. However, selecting appropriate hyperparameters and monitoring convergence is essential to ensure optimal performance.

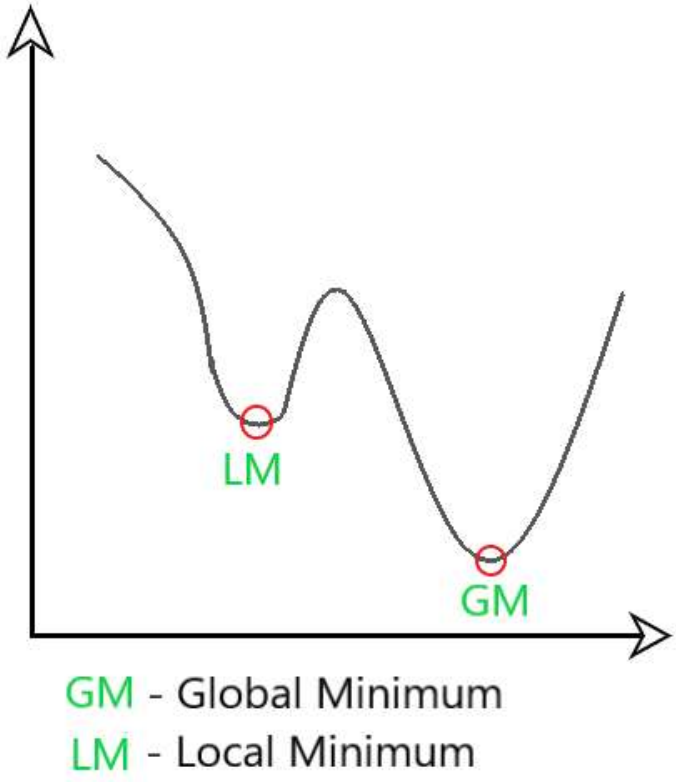


Fig 2.5.4: A Plot showing the global and local minimum. Adam's Optimizer finds the Global Minimum [9]

CHAPTER 3: REQUIREMENT ANALYSIS

3.1 Problem Definition

The objective of this project is to understand the emotion present in an audio file. This task had many problems which in turn had many sub-problems. The problems being:

1. The machine doesn't understand sound. So an audio file cannot be fed as data to a model to get results. It is very difficult to store so many audio files in a dataset. So, utilising the MFCC feature extraction technique to convert the raw audio signals into a set of representative features. Extracting MFCC features from each audio file in the dataset, capturing relevant information about the frequency distribution and dynamics of the speech signals.

2. Choosing an appropriate machine learning algorithm for emotion classification is a big problem. Then splitting the dataset into training and testing sets for model evaluation and validation. After which training the machine learning model using the extracted MFCC features as input and the labelled emotions as output. Lastly, optimisation of the model hyperparameters to improve performance and generalisation.

3. Evaluating the trained model's performance using appropriate evaluation metrics such as accuracy, precision, recall, and F1-score is the next problem. Evaluation methods need to be in place for this.

4. Ensuring the program provides clear and interpretable output, indicating the detected emotion(s) in the audio files is the next problem. Proper testing measures need to be taken to ensure that proper output is given by the model.

3.2 Software Requirement

1.	Operating System used:	Windows 11
2.	Programming languages used:	Python
3.	Python version used:	3.11.5
4.	IDE used:	Visual Studio Code, Kaggle Notebooks and Jupyter-Notebooks.
5.	Softwares used:	draw.io, Microsoft Office

6. Python Packages Used:
    a. For data visualisation:
        i. matplotlib
        ii. seaborn

    b. For mathematical processing:
        i. numpy

    c. For dataset preprocessing:
        i. pandas

    d. For audio processing and visualisation:
        i. librosa

    e. For modelling:
        i. sklearn
        ii. tensorflow
        iii. KNeighborsClassifier
        iv. LSTM
        v. SVC

    f. For data normalisation:
        i. StandardScaler

    g. For accuracy measurement and report:
        i. accuracy_score
        ii. classification_report

    h. For visualising training and validation accuracy:
        i. confusion_matrix
        ii. learning_curve

    i. For audio playback:
        i. IPython.display

    j. For system interactions and file management:
        i. os
        ii. sys
        iii. glob

    k. For model serialisation:
        i. pickle
        ii. joblib

    l. For progress bar:
        i. tqdm

    m. For handling warnings:
        i. warnings

3.3 Hardware Requirements

Device Specifications:

CPU: AMD Ryzen 3 3250U with Radeon Graphics @2.60 GHz
RAM: ~8GB
Disk: ~1TB HDD

For implementing the models of machine learning, Kaggle was used. Kaggle hardware specifications for the free version are:

Accelerator: GPU P100 (~15GB Available)
RAM: ~28GB Available
Disk: ~66GB Available

CHAPTER 4: SYSTEM DESIGN

4.1 Conceptual Models

4.1.1 Dataset Handling

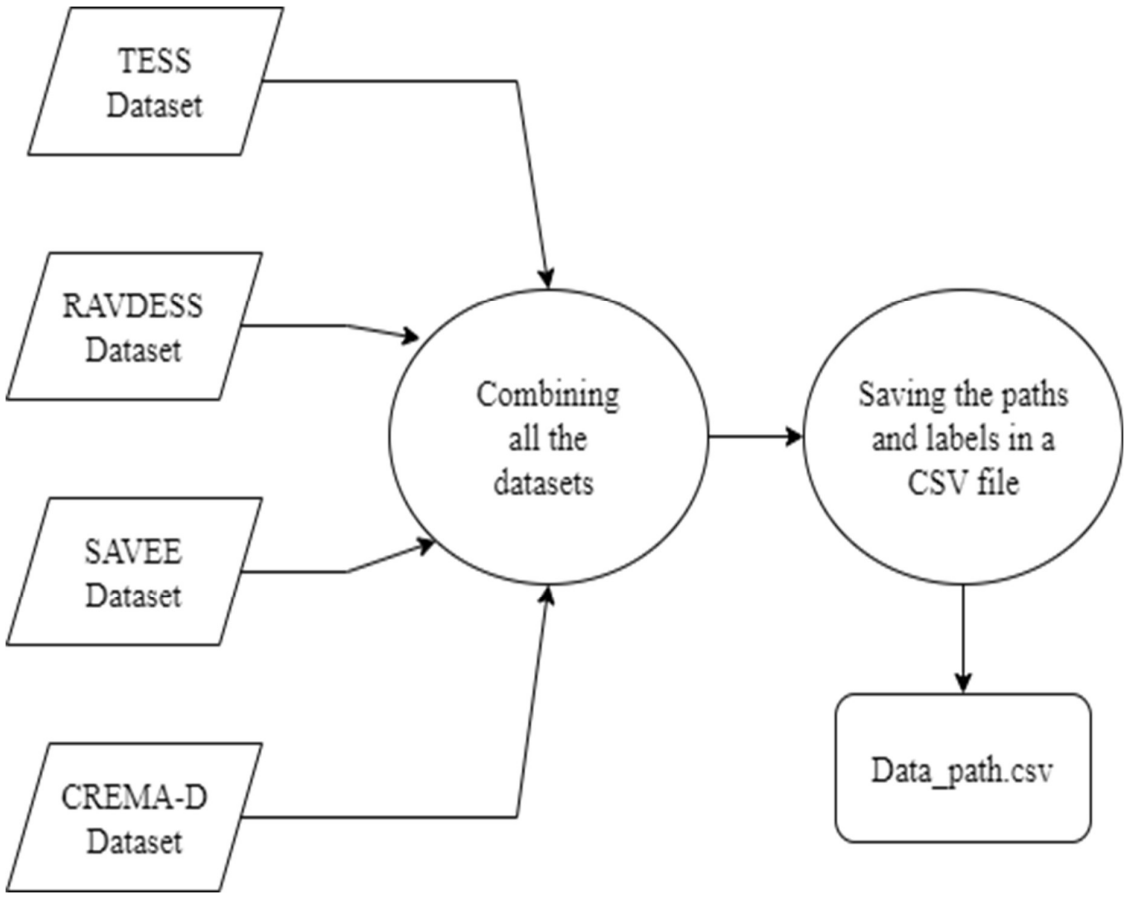


Fig 4.1.1: A flowchart showing the Dataset Handling

4.1.2 Feature Extraction Flowchart

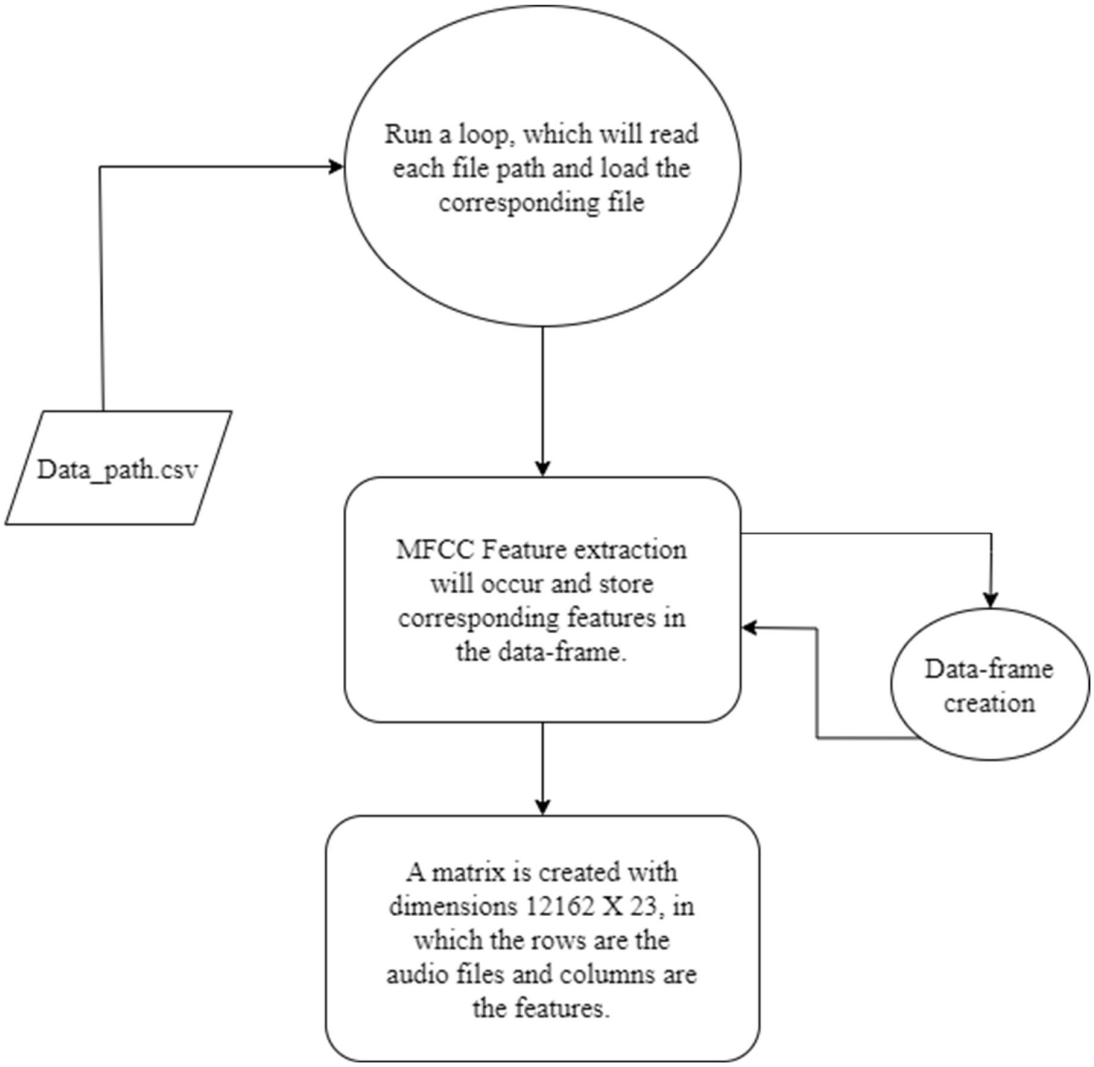


Fig 4.1.2: A flowchart showing the Feature Extraction

4.1.3 Model Training & Validation

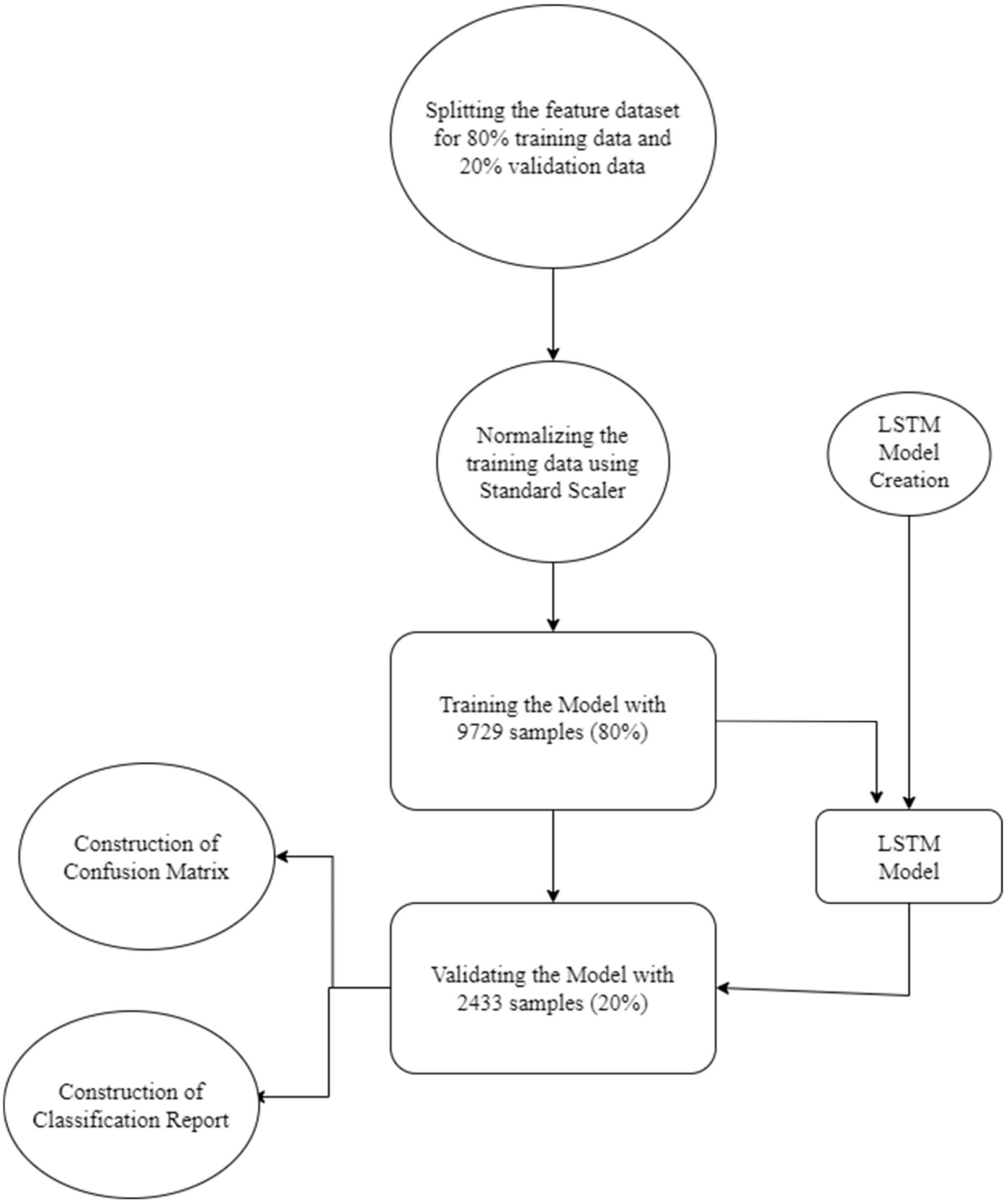


Fig 4.1.3: A flowchart showing the Model Creation, Training & Testing

4.1.4 Flowchart showing the Full Working

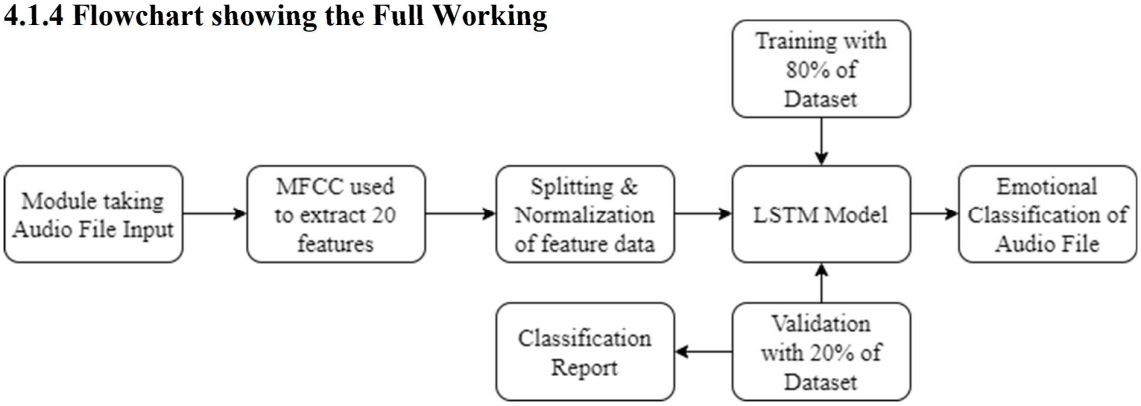


Fig 4.1.4: A flowchart showing the Full Working

4.2 Procedural Design

This section focuses on the various algorithms or pseudocodes that have been used in compiling the entire algorithm of the project.

```
# Read audio file paths from CSV file
file_paths = READ_CSV("file_path_containing_audio_paths")

# Set the number of MFCC coefficients and the number of audio files
num_mfcc = 20
num_files = 12162

# Create a list of column names for the DataFrame
columns = []
FOR i FROM 0 TO num_mfcc - 1
   column_name = "mfcc_" + (i + 1)
   APPEND column_name TO columns
END FOR

# Initialise a DataFrame with the created column names and row indices
df = CREATE_DATAFRAME(columns, index=RANGE(num_files))

# Create a progress bar
pbar = CREATE_PROGRESS_BAR(total=num_files)

# Loop through audio files and extract MFCC features
FOR idx, path IN ENUMERATE(file_paths.path)
      # Load the audio file and extract MFCC features
        audio_data, sample_rate = LOAD_AUDIO(parameters)
        mfccs = EXTRACT_MFCC(audio_data, sample_rate, num_mfcc)

      # Compute the mean of MFCC coefficients along each column
         mfccs_mean = COMPUTE_MEAN(mfccs, axis=1)
```

```
    # Store the mean coefficients in the DataFrame
      df.loc[idx] = mfccs_mean

END FOR
```

Algo 4.2.1: Feature Extraction using MFCC

```
# Reshape input data for LSTM
X_train_reshaped = RESHAPE(X_train_scaled, (X_train_scaled.SHAPE[0], 1,
X_train_scaled.SHAPE[1]))
X_test_reshaped = RESHAPE(X_test_scaled, (X_test_scaled.SHAPE[0], 1,
X_test_scaled.SHAPE[1]))

# One-hot encode the labels
label_binarizer = CREATE_LABEL_BINARIZER()
y_train_one_hot = label_binarizer.FIT_TRANSFORM(y_train)
y_test_one_hot = label_binarizer.TRANSFORM(y_test)
num_classes = 14

# Defining the LSTM model
lstm_model = CREATE_SEQUENTIAL_MODEL()
lstm_model.ADD(LSTM(X_train_reshaped)
lstm_model.ADD(DROPOUT(0.5))
lstm_model.ADD(DENSE(64, activation='relu'))
```

```
lstm_model.ADD(DENSE(num_classes, activation='softmax'))

# Compiling the model
lstm_model.COMPILE(learning_rate_parameters)

# Training the model
history = lstm_model.FIT(training_parameters)

# Evaluating the model and calculate accuracy
PRINT("Test Accuracy:", accuracy)
```

Algo 4.2.2: LSTM Model Creation

```
# Read the extracted features from a CSV file
extracted_features = READ_CSV("file_path_containing_features")

# Split the data into training and testing sets
X_data = extracted_features.DROP(['path', 'labels', 'source'], axis=1)
y_data = extracted_features['labels']
# Model training and validation
X_train, X_test, y_train, y_test = SPLIT_DATA(X_data, y_data, test_size=0.20,
shuffle=True, random_state=42)
```

Algo 4.2.3: LSTM Model Training and Testing

4.3 About the Datasets

4.3.1 SAVEE Dataset (Surrey Audio Visual Expressed Emotion)

The SAVEE database was recorded from four native English male speakers (identified as DC, JE, JK, KL), postgraduate students and researchers at the University of Surrey aged from 27 to 31 years. Emotion has been described psychologically in discrete categories: anger, disgust, fear, happiness, sadness and surprise. A neutral category is also added to provide recordings of 7 emotion categories.

The text material consisted of 15 TIMIT sentences per emotion: 3 common, 2 emotion-specific and 10 generic sentences that were different for each emotion and phonetically-balanced. The 3 common and 2 × 6 = 12 emotion-specific sentences were recorded as neutral to give 30 neutral sentences. This resulted in a total of 120 utterances per speaker

The audio files are named in such a way that the prefix letters describes the emotion classes as follows:

- ➔ 'a' = 'anger'
- ➔ 'd' = 'disgust'

- ➔ 'f' = 'fear'
- ➔ 'h' = 'happiness'
- ➔ 'n' = 'neutral'
- ➔ 'sa' = 'sadness'
- ➔ 'su' = 'surprise'

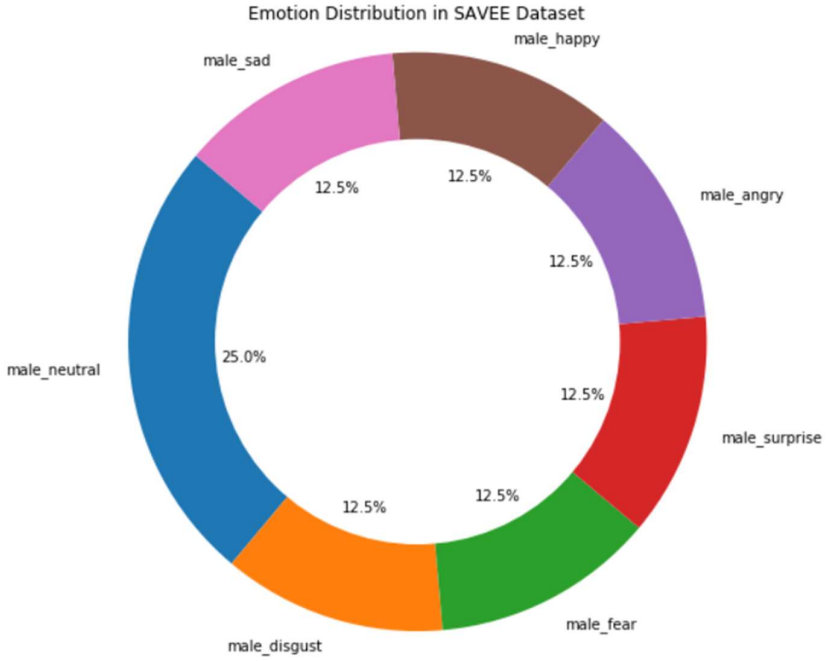


Fig 4.3.1: Emotion Distribution in SAVEE Dataset

4.3.2 RAVDESS Dataset (Ryerson Audio-Visual Database of Emotional Speech and Song)

RAVDESS contains 1440 files: 60 trials per actor x 24 actors = 1440. The RAVDESS contains 24 professional actors (12 female, 12 male), vocalising two lexically-matched statements in a neutral North American accent. Speech emotions include calm, happy, sad, angry, fearful, surprise, and disgust expressions. Each expression is produced at two levels of emotional intensity (normal, strong), with an additional neutral expression.

The filename identifiers as per the official RAVDESS website:

- ➔ Modality (01 = full-AV, 02 = video-only, 03 = audio-only).
- ➔ Vocal channel (01 = speech, 02 = song).
- ➔ Emotion (01 = neutral, 02 = calm, 03 = happy, 04 = sad, 05 = angry, 06 = fearful, 07 = disgust, 08 = surprised).
- ➔ Emotional intensity (01 = normal, 02 = strong). NOTE: There is no strong intensity for the 'neutral' emotion.
- ➔ Statement (01 = "Kids are talking by the door", 02 = "Dogs are sitting by the door").
- ➔ Repetition (01 = 1st repetition, 02 = 2nd repetition).

➔ Actor (01 to 24. Odd numbered actors are male, even numbered actors are female).

Database Citation: [4]

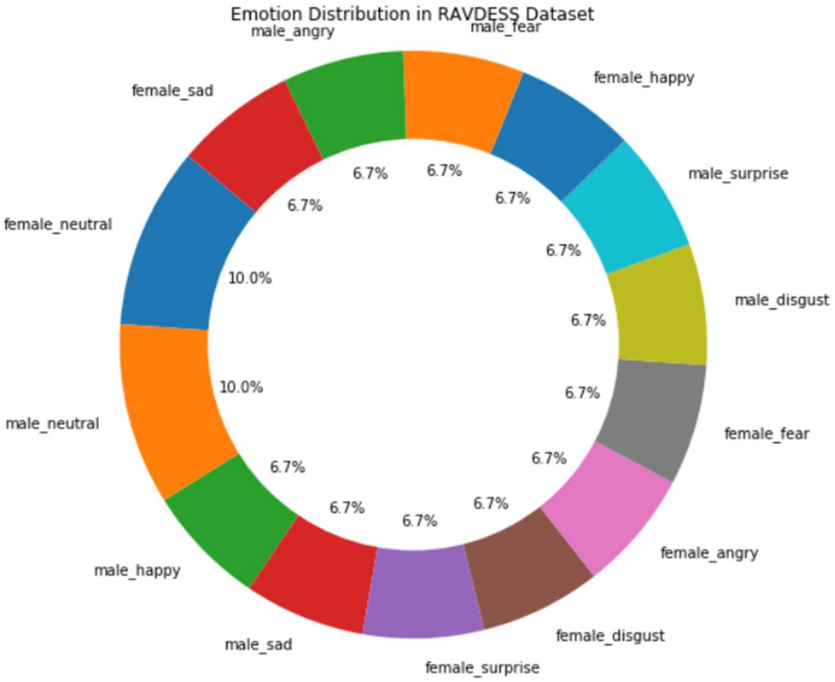


Fig 4.3.2: Emotion Distribution in RAVDESS Dataset

4.3.3 TESS Dataset (Toronto Emotional Speech Set)

A set of 200 target words were spoken in the carrier phrase "Say the word _' by two actresses (aged 26 and 64 years) and recordings were made of the set portraying each of seven emotions (anger, disgust, fear, happiness, pleasant surprise, sadness, and neutral). There are 2800 data points (audio files) in total.

The dataset is organised such that each of the two female actors and their emotions are contained within its own folder. And within that, all 200 target words audio files can be found. The format of the audio file is a WAV format.

Database Citation: [5]

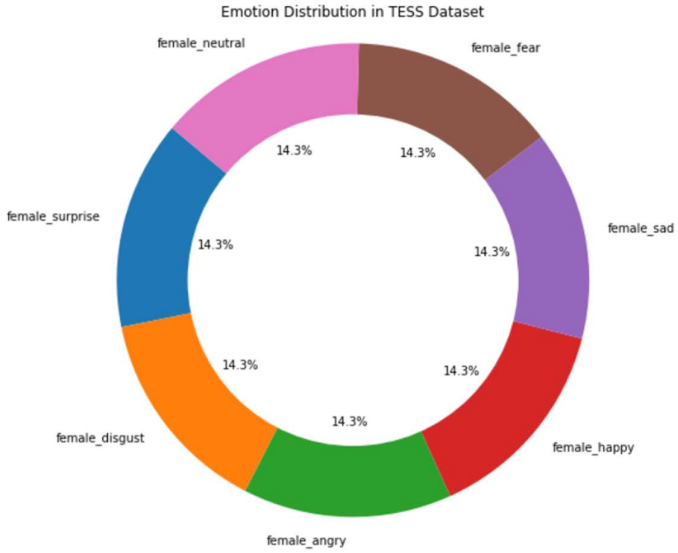


Fig 4.3.3: Emotion Distribution in TESS Dataset (on the right)

4.3.4 CREMA-D Dataset (Crowd Sourced Emotional Multimodal Actors Dataset)

CREMA-D is a data set of 7,442 original clips from 91 actors. These clips were from 48 male and 43 female actors between the ages of 20 and 74 coming from a variety of races and ethnicities (African American, Asian, Caucasian, Hispanic, and Unspecified). Actors spoke from a selection of 12 sentences. The sentences were presented using one of six different emotions (Anger, Disgust, Fear, Happy, Neutral, and Sad) and four different emotion levels (Low, Medium, High, and Unspecified).

Database Citation: [6]
Source of Information: Kaggle.

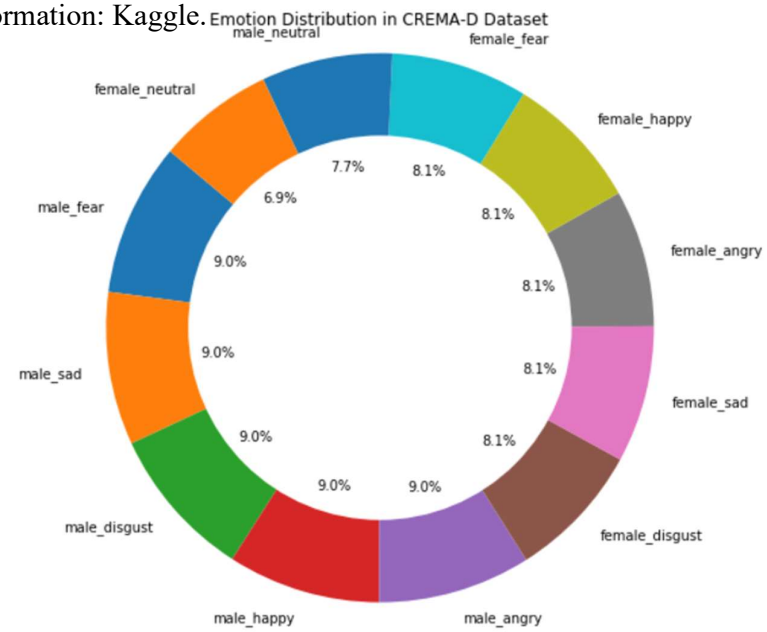


Fig 4.3.4: Emotion Distribution in CREMA-D Dataset

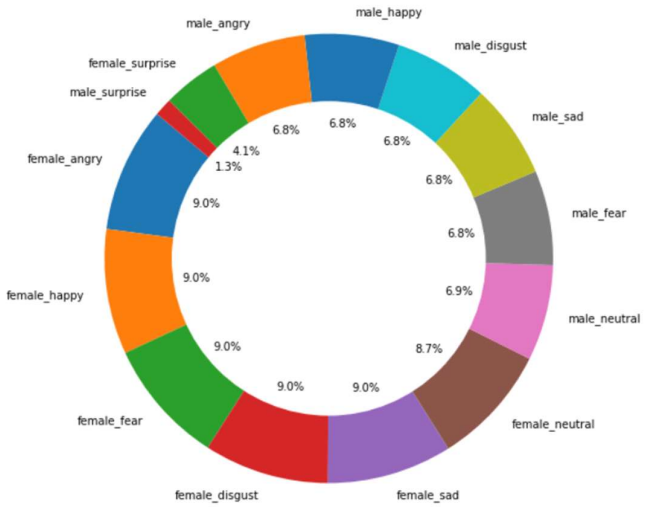


Fig 4.3.5: Emotion Distribution in the combined Dataset

CHAPTER 5: CODING AND TESTING

5.1 Coding Details

5.1.1 Importing Libraries

```
import librosa #used for audio processing tasks
import librosa.display #provides functions for visualizing audio data
import numpy as np #used for numerical computations
import matplotlib.pyplot as plt #used for creating static and interactive visualizations
from matplotlib.pyplot import specgram
import pandas as pd #package for data manipulation and library analysis
import glob #used for file operations and pathname pattern matching
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import learning_curve
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler #used for normalization of data
from sklearn.neighbors import KNeighborsClassifier #required for KNN model
from sklearn.svm import SVC #required for SVM model
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from sklearn.externals import joblib #used for model serialization
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam
import IPython.display as ipd #used for displaying audio signals
from tqdm import tqdm #progress bar
import seaborn as sns #used for visualization
import pickle #used for serialization and of Python objects
import os #used for file operations and manipulating environment variables
import sys

import warnings
# ignore warnings
if not sys.warnoptions:
    warnings.simplefilter("ignore")
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

The first step is to import various Python libraries and packages necessary for conducting sentiment analysis using Mel-frequency cepstral coefficients (MFCC) feature extraction and Modeling. Each library serves a specific purpose in the data preprocessing, feature extraction, model building, evaluation, and visualisation stages of the sentiment analysis pipeline.

The following are the explanations of some of the important packages:

1. librosa: This library is used for audio processing tasks, including loading audio files, extracting features such as MFCCs, and visualising audio data.
2. librosa.display: It provides functions for visualising audio data, which will be useful for displaying spectrograms and other audio representations.
3. numpy: Used for numerical computations, especially when manipulating arrays and matrices, which are essential for handling MFCCs and other numerical data.
4. matplotlib.pyplot: This library is widely used for creating static and interactive visualisations, including plotting graphs and charts.

5. pandas: A powerful package for data manipulation and analysis, often used for handling structured data and tabular datasets.
6. glob: This module is used for file operations and pathname pattern matching, allowing easy file management and iteration over files in directories.
7. sklearn.metrics: Provides various metrics for evaluating model performance, such as confusion matrix, accuracy score, and classification report.
8. sklearn.model_selection: Contains functions for splitting data into training and testing sets, as well as generating learning curves for model evaluation.
9. sklearn.preprocessing: Includes preprocessing techniques such as standardisation, which is often applied to scale features to a standard normal distribution.
10. sklearn.neighbors: Provides implementations of the k-nearest neighbours (KNN) classifier.
11. sklearn.svm: Contains implementations of support vector machine (SVM) classifiers.
12. tensorflow: It is an open-source machine learning framework developed by Google, widely used for building and training deep learning models.
13. tensorflow.keras.models.Sequential: Allows building sequential models in TensorFlow, which are neural network architectures where layers are stacked sequentially.
14. tensorflow.keras.layers: Provides various layers for constructing neural networks, such as LSTM (Long Short-Term Memory) and Dense layers.
15. tensorflow.keras.optimizers.Adam: An optimizer algorithm used for optimising the learning process during neural network training.
16. IPython.display: Provides utilities for displaying audio signals and playback.
17. tqdm: A library for adding progress bars to Python loops, enhancing the visual representation of the execution progress.
18. seaborn: Built on top of Matplotlib, Seaborn is used for creating visually appealing statistical graphics, which can be helpful for visualising model performance and data distributions.
19. pickle: Used for serialisation and deserialization of Python objects, allowing saving and loading of trained models and other data structures.
20. os: Provides functions for interacting with the operating system, such as file and directory operations, environment variable manipulation, etc.
21. sys: System-specific parameters and functions for Python runtime environment.

5.1.2 Dataset Handling

Four datasets are used which specialise for emotion classification. The four datasets are as follows:

- TESS (Toronto Emotional Speech Set)
- RAVDESS (Ryerson Audio-Visual Database of Emotional Speech and Song)
- SAVEE (Surrey Audio Visual Expressed Emotion)
- CREMA-D (Crowd Sourced Emotional Multimodal Actors Dataset)

The above datasets are slightly re-structured to satisfy the required folder structure.

```
# Traversing the directory tree rooted at the input directory
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        os.path.join(dirname, filename)

# Defining the paths to the datasets
TESS = "/kaggle/input/toronto-emotional-speech-set-tess/tess toronto emotional speech set data/TESS Toronto emotional speech set data/"
RAV = "/kaggle/input/ravdess-emotional-speech-audio/audio_speech_actors_01-24/"
SAVEE = "/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/"
CREMA = "/kaggle/input/cremad/AudioWAV/"
```

The os.walk() function is used to traverse the directory tree rooted at the specified path (/kaggle/input). It retrieves a tuple comprising the directory path, a list of subdirectories, and a list of filenames within the traversed directory. Utilising os.path.join(), it iterates through each directory and filename to acquire the full file paths.

5.1.3 Data Pre-Processing and Dataset Exploring

```
dir_list = os.listdir(RAV)
dir_list.sort()

# List initialization to store emotions and file paths
emotion = []
gender = []
path = []
for i in dir_list:
    fname = os.listdir(RAV + i)
    for f in fname:
        part = f.split('.')[0].split('-')
        emotion.append(int(part[2]))
        temp = int(part[6])
        if temp%2 == 0:
            temp = "female"
        else:
            temp = "male"
        gender.append(temp)
        path.append(RAV + i + '/' + f)

# Dataframe creation to store emotion labels, file paths and source dataset
RAV_df = pd.DataFrame(emotion)
RAV_df = RAV_df.replace({1:'neutral', 2:'neutral', 3:'happy', 4:'sad',
                         5:'angry', 6:'fear', 7:'disgust', 8:'surprise'})
RAV_df = pd.concat([pd.DataFrame(gender),RAV_df],axis=1)
RAV_df.columns = ['gender','emotion']
RAV_df['labels'] =RAV_df.gender + '_' + RAV_df.emotion
RAV_df['source'] = 'RAVDESS'
RAV_df = pd.concat([RAV_df,pd.DataFrame(path, columns = ['path'])],axis=1)
RAV_df = RAV_df.drop(['gender', 'emotion'], axis=1)
print(RAV_df.labels.value_counts())
print() # for extra new line
```

This section covers data preprocessing and exploration of the four datasets. It involves navigating the directory structure of the datasets to access audio files stored in various subdirectories.

```
dir_list = os.listdir(TESS)
dir_list.sort()

# List initialization to store emotions and file paths
path = []
emotion = []

for i in dir_list:
    fname = os.listdir(TESS + i)
    for f in fname:
        if i == 'OAF_angry' or i == 'YAF_angry':
            emotion.append('female_angry')
        elif i == 'OAF_disgust' or i == 'YAF_disgust':
            emotion.append('female_disgust')
        elif i == 'OAF_Fear' or i == 'YAF_fear':
            emotion.append('female_fear')
        elif i == 'OAF_happy' or i == 'YAF_happy':
            emotion.append('female_happy')
        elif i == 'OAF_neutral' or i == 'YAF_neutral':
            emotion.append('female_neutral')
        elif i == 'OAF_Pleasant_surprise' or i == 'YAF_pleasant_surprised':
            emotion.append('female_surprise')
        elif i == 'OAF_Sad' or i == 'YAF_sad':
            emotion.append('female_sad')
        else:
            emotion.append('Unknown')
        path.append(TESS + i + "/" + f)

# Dataframe creation to store emotion labels, file paths and source dataset
TESS_df = pd.DataFrame(emotion, columns = ['labels'])
TESS_df['source'] = 'TESS'
TESS_df = pd.concat([TESS_df,pd.DataFrame(path, columns = ['path'])],axis=1)
print(TESS_df.labels.value_counts())
print() # for an extra line
```

```
dir_list = os.listdir(CREMA)
dir_list.sort()

# List initialization to store emotions and file paths
gender = []
emotion = []
path = []
female = [1002,1003,1004,1006,1007,1008,1009,1010,1012,1013,1018,1020,1021,1024,1025,1028,
          1029,1030,1037,1043,1046,1047,1049,1052, 1053,1054,1055,1056,1058,1060,1061,1063, 1072,1073,1074,1075,1076,1078,1079,1082,
          1084,1089,1091]

for i in dir_list:
    part = i.split('_')
    if int(part[0]) in female:
        temp = 'female'
    else:
        temp = 'male'
    gender.append(temp)
    if part[2] == 'SAD' and temp == 'male':
        emotion.append('male_sad')
    elif part[2] == 'ANG' and temp == 'male':
        emotion.append('male_angry')
    elif part[2] == 'DIS' and temp == 'male':
        emotion.append('male_disgust')
    elif part[2] == 'FEA' and temp == 'male':
        emotion.append('male_fear')
    elif part[2] == 'HAP' and temp == 'male':
        emotion.append('male_happy')
    elif part[2] == 'NEU' and temp == 'male':
        emotion.append('male_neutral')
    elif part[2] == 'SAD' and temp == 'female':
        emotion.append('female_sad')
    elif part[2] == 'ANG' and temp == 'female':
        emotion.append('female_angry')
    elif part[2] == 'DIS' and temp == 'female':
        emotion.append('female_disgust')
    elif part[2] == 'FEA' and temp == 'female':
        emotion.append('female_fear')
    elif part[2] == 'HAP' and temp == 'female':
        emotion.append('female_happy')
    elif part[2] == 'NEU' and temp == 'female':
        emotion.append('female_neutral')
    else:
        emotion.append('Unknown')
    path.append(CREMA + i)


#Dataframe creation to store emotion labels, file paths and source dataset
CREMA_df = pd.DataFrame(emotion, columns = ['labels'])
CREMA_df['source'] = 'CREMA'
CREMA_df = pd.concat([CREMA_df,pd.DataFrame(path, columns = ['path'])],axis=1)
print(CREMA_df.labels.value_counts())
print() # for an extra line
```

```
dir_list = os.listdir(SAVEE)

# List initialization to store emotions and file paths
emotion =[]
path = []

for i in dir_list:
    if i[-8:-6]=='_a':
        emotion.append('male_angry')
    elif i[-8:-6]=='_d':
        emotion.append('male_disgust')
    elif i[-8:-6]=='_f':
        emotion.append('male_fear')
    elif i[-8:-6]=='_h':
        emotion.append('male_happy')
    elif i[-8:-6]=='_n':
        emotion.append('male_neutral')
    elif i[-8:-6]=='sa':
        emotion.append('male_sad')
    elif i[-8:-6]=='su':
        emotion.append('male_surprise')
    else:
        emotion.append('male_error')
    path.append(SAVEE + i)

# Dataframe creation to store emotion labels, file paths and source dataset
SAVEE_df = pd.DataFrame(emotion, columns = ['labels'])
SAVEE_df['source'] = 'SAVEE'
SAVEE_df = pd.concat([SAVEE_df, pd.DataFrame(path, columns = ['path'])], axis = 1)
print(SAVEE_df.labels.value_counts())
print() #for an extra line
```

Next, the emotion labels are extracted from the filenames using a predefined naming convention within the dataset. These labels encompass emotions categorised as male and female, including anger, disgust, fear, happiness, neutral, sadness, and surprise. We then generate full file paths for each audio sample by combining the dataset directory path with the corresponding filename. This process ensures that each audio sample is associated with its respective emotion label, enabling effective analysis and classification during subsequent stages of the project.

Once the dataset is constructed, incorporating emotion labels, file paths, and data sources, it proceeds with exploratory data analysis. This involves assessing the distribution of emotion labels across the datasets. By analysing the balance of emotion classes and the overall composition of the dataset, valuable insights into the characteristics of the data is gained. This exploratory phase aids in understanding the prevalence of different emotions and ensures that the dataset is adequately representative of the emotions under consideration. Such insights guide further preprocessing steps and model development, enhancing the effectiveness of subsequent analyses and classification tasks.

5.1.4 Data-frames Handling

In total, from the four datasets we have 12,162 audio samples. Code Snippet on the next page.

```
# Concatenate dataframes from different emotion datasets into a single dataframe
df = pd.concat([SAVEE_df, RAV_df, TESS_df, CREMA_df], axis = 0)

# Count the occurrences of each audio label
audio_label_count = df.labels.value_counts()
print(audio_label_count)
print(df.to_csv("Data_path.csv",index=False)) # Saving the combined dataframe to a CSV file


# Generating bar plot
colors = ['blue', 'green', 'red', 'purple', 'orange',
          'yellow', 'pink', 'brown', 'gray', 'cyan',
          'magenta', 'olive', 'teal', 'navy']
pastel_colors = sns.color_palette("muted", len(colors))

plt.figure(figsize=(22, 10))
plt.bar(audio_label_count.index, audio_label_count.values, color=pastel_colors)
plt.xlabel('Label')
plt.ylabel('Count')
plt.title('Audio Label Count')
plt.xticks(audio_label_count.index)
plt.show()
```

In this section, it merges the data frames derived from four distinct emotion datasets, namely SAVEE, RAVDESS, TESS, and CREMA-D, into a unified dataframe named 'df'. A data frame is a tabular data structure commonly used in data analysis, where rows represent individual observations (in this case, audio samples) and columns represent variables (such as emotion labels, file paths, etc.). By combining these separate data frames, it consolidates the information from all datasets into a single coherent dataset, facilitating streamlined analysis and manipulation.

Once the consolidation is complete, it calculates the count of audio samples corresponding to each emotion label present in the 'df' dataframe. This step allows gaining insights into the distribution of emotions across the entire dataset, identifying any potential imbalances or biases that may exist. To visually represent this distribution, it utilises a bar plot, where each emotion label is depicted along the x-axis, and the corresponding count of audio samples is displayed along the y-axis. This graphical representation offers a clear and intuitive understanding of the relative frequency of different emotions within the dataset.

Furthermore, to ensure the preservation and accessibility of the consolidated dataset for future reference and analysis, it saves the combined dataframe 'df' to a CSV (Comma-Separated Values) file. This file format is widely used for storing tabular data in a plain text format, making it easily readable and compatible with various data analysis tools and software. By saving the data frame to a CSV file, it establishes a durable and portable data storage solution, enabling seamless sharing, collaboration, and reproducibility of analyses across different platforms and environments.

5.1.5 Feature Extraction

The MFCC Feature Extraction Method is used in this project. The complete data processing took around 11 minutes as it is iterating over 4 datasets.

```
ref = pd.read_csv("/kaggle/working/Data_path.csv")
num_mfcc = 20  # Number of MFCC coefficients
num_files = 12162  # Number of audio files

# Dataframe initialization
columns = ['mfcc_' + str(i+1) for i in range(num_mfcc)]
df = pd.DataFrame(columns=columns, index=range(num_files))
pbar = tqdm(total=num_files) # Progress bar

# Loop through audio files and extract MFCC features
for idx, path in enumerate(ref.path):
    #Loading the audio file from the specified path and extracting MFCC features
    X, sample_rate = librosa.load(path, res_type='kaiser_fast', duration=2.5, sr=44100, offset=0.5)
    mfccs = librosa.feature.mfcc(y=X, sr=sample_rate, n_mfcc=num_mfcc)

    mfccs_mean = np.mean(mfccs, axis=1)  # Compute mean of MFCC coefficients along each column
    df.loc[idx] = mfccs_mean # Storing the mean coefficients in the dataframe
    pbar.update(1)

pbar.close()
```

Within this section, it begins by importing the CSV file that it previously saved. This file contains the paths to audio files sourced from various emotion datasets. Next, it proceeds to extract Mel-frequency cepstral coefficients (MFCC) features from each audio file. MFCC features are commonly used in audio signal processing and are effective in capturing the spectral characteristics of audio signals.

Once the MFCC features are extracted, it computes the mean of the coefficients along each column, resulting in a feature vector for each audio file. These feature vectors are then stored in a dataframe. This data frame serves as the input features for training machine learning models aimed at sentiment analysis. By aggregating the MFCC coefficients and computing their means, it creates a concise representation of each audio sample that captures essential information about its spectral content. This dataframe of feature vectors enables efficient training of machine learning models, allowing them to learn patterns and associations between audio features and emotional sentiment.

The MFCC Extraction process is performed as follows:

1. It loads the audio file from the specified path, opting for 'kaiser_fest' as the resampling method to expedite loading.
2. It sets the sampling rate to 44.1 kHz.
3. We decided to utilise 20 MFCC features after assessing various options. We acknowledge that while additional features may enhance interpretation, they could also elevate the likelihood of overfitting.
4. It computes the mean of the MFCC coefficients across each column.
5. A progress bar is integrated to monitor the extraction process.

```
print(df.shape)
#Concatenating the two dataframes to match the mfcc features
#of a particular audio file to its emotion label
df_concat = pd.concat([ref, df], axis=1)
# Filling any missing values(NaN) in the dataframe with zero.
# This is done for easier handling of missing data.
df_concat=df_concat.fillna(0)
df_concat.to_csv('features.csv', index=False)
print(df_concat.shape)
df_concat[:5]
```

Concatenation of ref and df DataFrames horizontally is done using pd.concat(). The resulting DataFrame df_concat has a shape of (12162, 23), indicating that it has retained the 12162 rows from both DataFrames but with 23 columns in total. The additional columns are from the ref DataFrame which has the 'emotion_label', 'source', 'path'. Any missing values (NaN) in the df_concat DataFrame are filled with zero using the fillna(0) method. This ensures that all missing data are handled uniformly for easier processing. Finally, the resulting DataFrame is saved to a CSV file named 'features.csv' using the to_csv() method.

5.1.6 Splitting of Training and Testing Datasets

```
extracted_features = pd.read_csv("/kaggle/working/features.csv")

X_train, X_test, y_train, y_test = train_test_split(extracted_features.drop(['path','labels','source'],
                                                                            axis=1)
                                                    , df_concat.labels
                                                    , test_size=0.20
                                                    , shuffle=True
                                                    , random_state=42 # Random seed for reproducibility
                                                   )

print(X_train.shape)
print(X_test.shape)
```

Now one of the crucial steps for machine learning is done here. The complete dataset is splitted into two sets, training and testing. The dataset is organised as a DataFrame, where each row represents an audio sample, and the columns represent various features extracted from the audio data. The columns 'path', 'labels', and 'source' are dropped from the feature matrix X, as they contain metadata and are not relevant for modelling. The resulting feature matrix contains only the extracted features necessary for training the model. The feature matrix X serves as the input data, while the 'labels' column from df_concat is designated as the target variable y, representing the labels for classification. The test_size parameter is set to 0.20, indicating that 20% of the data will be reserved for testing, while the remaining 80% will be used for training. The data is randomly shuffled before splitting, to prevent any inherent ordering bias in the dataset. A specific random seed is also set to ensure reproducibility across multiple runs to ensure that the same data split is obtained each time the code is executed.

5.1.7 Z-score Normalisation

```
# Normalization of training and testing data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

The provided code snippet implements standardisation for both the training and testing datasets. Standardisation, also referred to as Z-score normalisation, aims to centre the features around a mean of 0 and scale them to have a standard deviation of 1. This process facilitates bringing all features to a comparable scale, thereby mitigating the risk of any individual feature exerting undue influence on the learning process due to its larger magnitude. By standardising the features, it ensures that each feature contributes proportionately to the model's learning without being disproportionately affected by its scale. This enhances the stability and performance of the machine learning models trained on the standardised data, leading to more reliable and interpretable results during subsequent analysis and inference tasks.

First, an instance of the StandardScaler() class is created, which will be used to standardise the data. Then, the fit_transform() method is applied to the training data (X_train), which calculates the mean and standard deviation of each feature in the training dataset and transforms the data accordingly. This step is crucial as it ensures that the mean of each feature becomes 0 and the standard deviation becomes 1.

By standardising the data in this manner, it ensures consistency in feature scaling across all dimensions, thereby preventing any single feature from disproportionately influencing the learning process. Consequently, models trained on standardised data are better equipped to generalise to unseen data and produce more reliable predictions.

5.1.8 KNN Model Training and Validation

```
#KNN classifier
knn = KNeighborsClassifier(n_neighbors=5, weights='uniform', metric='euclidean')

# Training
knn.fit(X_train_scaled, y_train)

# Predict labels
y_pred = knn.predict(X_test_scaled)

# Evaluating the classifier and calculating accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Initially, we employed a K-Nearest Neighbors (KNN) classifier for classification tasks, leveraging the scikit-learn library. This classifier assigns labels to new data points based on the majority class among their k nearest neighbours in the feature space. By considering the

proximity of data points in the feature space, KNN enables intuitive and effective classification, making it a popular choice for various machine learning applications.

An instance of the KNN classifier is instantiated with specific parameters. These parameters include n_neighbors, which determines the number of neighbours considered when making predictions, weights to specify the weight function used in prediction, and metric to define the distance metric used for calculating the distance between instances. In this case, the n_neighbors parameter is set to 5, 'uniform' weighting is applied to consider all neighbours equally, and 'euclidean' distance is used as the metric.

Next, the classifier is trained on the scaled training data using the fit method. The scaled training features (X_train_scaled) and their corresponding labels (y_train) are provided as input to the fit method to train the model. Once trained, the classifier proceeds to make predictions on the scaled testing data (X_test_scaled) using the predict method. The predicted labels are stored in the variable y_pred.

Subsequently, the accuracy of the classifier is evaluated by comparing the predicted labels (y_pred) with the true labels (y_test) using the accuracy_score function. The accuracy score represents the proportion of correctly classified instances in the test set, providing insights into the classifier's performance.

```
def plot_confusion_matrix(model, X, y, display_labels=None, cmap=plt.cm.Blues):
    y_pred = model.predict(X)
    cm = confusion_matrix(y, y_pred)

    if display_labels is None:
        display_labels = np.unique(y)

    # Plotting
    plt.figure(figsize=(10, 8))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title('Confusion Matrix')
    plt.colorbar()
    tick_marks = np.arange(len(display_labels))
    plt.xticks(tick_marks, display_labels, rotation=45)
    plt.yticks(tick_marks, display_labels)


    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, format(cm[i, j], 'd'),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

The function plot_confusion_matrix is a utility designed to visualise the confusion matrix, a fundamental tool for evaluating the performance of classification models. It serves to provide a comprehensive understanding of how well a model performs across different classes by comparing its predicted labels with the true labels of a dataset. The function accepts several

parameters, including the trained classification model (model), the feature matrix (X), and the true labels (y). Additionally, it allows for customization of the displayed labels (display_labels) and the colormap (cmap) used in the plot.

Upon invocation, the function first generates predictions (y_pred) using the provided model and feature matrix. Subsequently, it calculates the confusion matrix (cm) by comparing the predicted labels with the true labels. Optionally, if display_labels is not specified, the function automatically determines unique labels from the true labels (y).

At the heart of the function's functionality is its capacity to generate a visual depiction of the confusion matrix through Matplotlib. Each cell within the matrix represents the number of instances where a specific true label intersects with a predicted label. The intensity of colour in each cell corresponds to the count, with darker hues indicating higher counts. To enhance comprehension, the function annotates each cell with its count, dynamically adjusting the text colour based on the intensity of the cell. Moreover, it incorporates a colorbar adjacent to the matrix, offering a visual aid for interpreting the counts within the cells. By providing this comprehensive visual representation, the function facilitates a clear understanding of the classification performance, enabling users to discern patterns and assess the model's accuracy and efficacy. This visualisation tool is instrumental in evaluating the performance of classification algorithms and identifying areas for improvement, thereby supporting informed decision-making in machine learning tasks.

Finally, the function adjusts the layout, sets axis labels, and displays the plot, offering a clear and intuitive depiction of the model's classification performance.

```
def plot_learning_curve(estimator, X, y, train_sizes=np.linspace(0.1, 1.0, 5), cv=5):
    train_sizes, train_scores, test_scores = learning_curve(estimator, X, y, train_sizes=train_sizes, cv=cv)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.title("Learning Curve")
    plt.xlabel("Training examples")
    plt.ylabel("Score")
    plt.grid()
    plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_mean + train_scores_std, alpha=0.1, color="r")
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std, test_scores_mean + test_scores_std, alpha=0.1, color="g")
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validation score")
    plt.legend(loc="best")
    plt.show()
```

The function plot_learning_curve is a tool designed to visualise the learning curve of the machine learning model. This curve provides valuable insights into the performance of the model as the size of the training dataset increases. The function accepts several parameters, including the estimator, the feature matrix (X), the target variable (y), and optional parameters such as train_sizes and cv.

Upon invocation, the function utilises the learning_curve function from scikit-learn to compute the training and cross-validation scores across different training set sizes. It does so by incrementally increasing the size of the training dataset from a specified range of

percentages (train_sizes) of the total dataset. Cross-validation (cv) is employed to evaluate the model's performance at each training set size.

The scores obtained are averaged to compute the mean and standard deviation for both training and cross-validation. These aggregated scores offer insights into the model's performance and its consistency across different training set sizes, serving as indicators of its variability and effectiveness in handling various data samples.

The core functionality of the function lies in its ability to plot the learning curve using Matplotlib. The plot displays the training and cross-validation scores against the number of training examples. The shaded regions around the mean scores represent the variability or standard deviation of the scores. The red shading indicates the variability in the training scores, while the green shading represents the variability in the cross-validation scores. The function also labels the axes, sets the title, and adds a legend to the plot for clarity.

```
plot_confusion_matrix(knn, X_test_scaled, y_test)
plot_learning_curve(knn, X_train_scaled, y_train)
```

```
# Saving the trained model to disk
joblib.dump(knn, 'knn_model.pkl')
print("Model saved at: /kaggle/working/")
```

It utilises the joblib.dump() function for serialisation, which involves saving the model object (KNN) and specifying the filename ('knn_model.pkl') where the serialised model will be stored. This function requires two arguments: the model object itself, representing the trained K-Nearest Neighbors classifier, and the designated filename for the saved model. By employing serialisation, we persist the trained model in a binary format, enabling easy storage and future retrieval. This process ensures that the model can be seamlessly reused or deployed in different environments without the need for retraining. The serialised model file, stored as 'knn_model.pkl', encapsulates the learned parameters and attributes of the KNN classifier, allowing for efficient sharing, distribution, and integration into other applications or workflows.

5.1.9 SVM Model Training and Validation

Next a Support Vector Machine (SVM) classifier is trained for a classification task using scikit-learn. Initially, the feature data, represented by X_train and X_test, undergoes conversion into NumPy arrays to facilitate subsequent processing. Given that SVM classifiers typically require input data in a 2D format, the arrays are reshaped if necessary using the reshape() function to ensure compatibility with the classifier's input requirements. This

reshaping ensures that the data is structured as a 2D array with dimensions corresponding to the number of samples and features.

```
X_train_array = np.array(X_train)
X_test_array = np.array(X_test)

# Reshape the array if necessary
X_train_flattened = X_train_array.reshape(X_train_array.shape[0], -1)
X_test_flattened = X_test_array.reshape(X_test_array.shape[0], -1)

# Initialize SVM classifier
svm_classifier = SVC(kernel='linear')

# Train the SVM classifier
svm_classifier.fit(X_train_flattened, y_train)

# Predict labels for test data
y_pred_svm = svm_classifier.predict(X_test_flattened)

# Calculate accuracy
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print("Accuracy:", accuracy_svm)
```

Following data preprocessing, an SVM classifier instance is instantiated and configured with a linear kernel by specifying kernel='linear'. This choice indicates that the classifier will employ a linear decision boundary to separate the classes within the feature space. Subsequently, the classifier is trained on the training data (X_train_flattened and y_train) using the fit() method. Here, X_train_flattened contains flattened feature vectors of the training samples, while y_train holds the corresponding target labels.

Once trained, the SVM classifier is employed to predict labels for the test data (X_test_flattened) via the predict() method. The predicted labels are stored in the variable y_pred_svm. To assess the performance of the classifier, its accuracy on the test data is computed using the accuracy_score() function. This involves comparing the predicted labels (y_pred_svm) against the true labels (y_test). Finally, the resulting accuracy score is calculated.

```
conf_matrix_svm = confusion_matrix(y_test, y_pred_svm)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_svm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
joblib.dump(svm_classifier, 'svm_model.pkl')
print("Model saved at /kaggle/working/")
```

The Confusion Matrix for the SVM classifier is thus generated to evaluate the accuracy of the model in classifying different emotion labels.

5.1.10 LSTM Model Training and Validation

This following code snippet outlines the construction, training, and evaluation of a Long Short-Term Memory (LSTM) neural network model designed for emotion classification tasks utilising Mel-frequency cepstral coefficients (MFCC) extracted from audio data. Initially, the input data, comprising the MFCC features, is reshaped to align with the LSTM model's input structure, conforming to the shape requirements of (batch_size, timesteps, features). Subsequently, the labels associated with the data are one-hot encoded using the LabelBinarizer from scikit-learn, enabling effective handling of multi-class classification scenarios. The LSTM model architecture is then specified, employing the Keras Sequential API. This architecture encompasses an LSTM layer with 128 units, followed by a dropout layer with a dropout rate of 0.5 to mitigate overfitting. Subsequently, a dense layer comprising 64 units with ReLU activation is incorporated, concluding with an output layer employing the softmax activation function, which suits multi-class classification objectives.

```
# Reshape input data for LSTM
X_train_reshaped = X_train_scaled.reshape(X_train_scaled.shape[0], 1, X_train_scaled.shape[1])
X_test_reshaped = X_test_scaled.reshape(X_test_scaled.shape[0], 1, X_test_scaled.shape[1])

# One-hot encode the labels
label_binarizer = LabelBinarizer()
y_train_one_hot = label_binarizer.fit_transform(y_train)
y_test_one_hot = label_binarizer.transform(y_test)
num_classes = 14

# Defining the LSTM model
lstm_model = Sequential([
    LSTM(units=128, input_shape=(X_train_reshaped.shape[1], X_train_reshaped.shape[2])),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(num_classes, activation='softmax')  # Use 'softmax' activation for multi-class classification
])

# Compiling the model
lstm_model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])

# Training the model
history = lstm_model.fit(X_train_reshaped, y_train_one_hot, epochs=100, batch_size=16, validation_data=(X_test_reshaped, y_test_one_hot))

# Evaluating the model and calculate accuracy
loss, accuracy = lstm_model.evaluate(X_test_reshaped, y_test_one_hot)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

One-hot encoding is a technique used in machine learning to convert categorical variables into a numerical format that can be used by algorithms for training and prediction. It involves representing each category as a binary vector, where each element corresponds to a unique category. The length of the vector is equal to the total number of unique categories in the variable. During encoding, a value of 1 is assigned to the element corresponding to the category, while all other elements are set to 0. This ensures that each category is represented distinctly and independently of others. One-hot encoding is preferred over label encoding for categorical variables that do not have an ordinal relationship, as it prevents the model from misinterpreting the categories as having a meaningful order. This encoding scheme is widely used in various machine learning tasks, including classification and natural language processing, to handle categorical data effectively.

After defining the model architecture, the next phase involves compilation, where the model is configured for training. This involves specifying the optimization algorithm, which in this case is the Adam optimizer with a learning rate of 0.001, and the loss function, which is categorical cross entropy. Additionally, the model's performance during training is monitored using accuracy as the metric.

Once compiled, the model is trained using the fit() method. During training, the optimization algorithm adjusts the model's parameters using the training data, while the validation data is used to evaluate the model's performance after each epoch. This iterative process continues for 100 epochs, with a batch size of 16 samples per iteration. This setup ensures efficient convergence towards the optimal model parameters, balancing between computational efficiency and model performance.

Following training, the model's efficacy is evaluated using the test dataset. The test loss and accuracy metrics are computed based on the model's predictions on the unseen test data. These metrics provide insights into the model's generalisation performance on new, unseen data, allowing for an assessment of its effectiveness in real-world scenarios. Overall, this comprehensive training and evaluation process ensures that the model is effectively trained and validated, ready for deployment in practical applications.

```
# Predict probabilities for the test set
y_pred_probs = lstm_model.predict(X_test_reshaped)

# Convert probabilities to class labels
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test_one_hot, axis=1)

# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, cmap='Blues', fmt='d', xticklabels=label_binarizer.classes_, yticklabels=label_binarizer.classes_)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

The confusion matrix for the predictions made by the LSTM model on the test set is computed and visualised.

First, the probabilities of each class label for the samples in the test set (X_test_reshaped) are predicted using the trained LSTM model (lstm_model.predict). These predicted probabilities (y_pred_probs) are then converted to class labels by selecting the index of the highest probability for each sample using the argmax() function along axis 1.

Similarly, the true class labels for the test set (y_true) are obtained by converting the one-hot encoded labels (y_test_one_hot) to categorical format using the argmax() function.

Next, the confusion matrix is computed using the confusion_matrix() function from scikit-learn's metrics module. The confusion matrix provides a summary of the model's

performance by tabulating the number of true positive, true negative, false positive, and false negative predictions for each class.

Finally, the confusion matrix is visualised using a heatmap, where each cell represents the number of samples that were classified into a particular predicted class versus their true class. The x-axis and y-axis of the heatmap correspond to the predicted and true class labels, respectively, and the cell values are annotated within the heatmap for clarity. The colormap used for the heatmap is set to 'Blues', which provides a visual representation of the distribution of correct and incorrect predictions across different classes.

```
# Plot learning and validation curves on the same graph
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.plot(history.history['acc'], label='Training Accuracy')
plt.plot(history.history['val_acc'], label='Validation Accuracy')
plt.title('Learning and Validation Curves')
plt.xlabel('Epochs')
plt.ylabel('Loss / Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

A plot showing the learning and validation curves of a neural network model trained using the LSTM architecture is displayed.

The learning and validation curves depict how the loss and accuracy of the model change over epochs during the training process. The x-axis of the plot represents the number of epochs, which are the iterations over the entire training dataset. The y-axis represents either the loss (measured by categorical cross-entropy) or accuracy. The lines in the plot show the values of the loss and accuracy metrics on the training and validation sets over epochs. The 'Training Loss' and 'Validation Loss' lines indicate the loss values obtained during training and validation, respectively, while the 'Training Accuracy' and 'Validation Accuracy' lines represent the corresponding accuracy values.

The learning curve illustrates the performance of the model on the training data, while the validation curve provides insights into the model's generalisation performance on unseen data.

In an ideal scenario, both the training and validation loss should decrease steadily over epochs, while the accuracy should increase. However, large gaps between the training and validation curves, or an increasing validation loss, may indicate overfitting. Conversely, if the training and validation curves converge to similar values, it suggests good model generalisation.

This visualisation helps in monitoring the training progress, detecting overfitting, and making decisions regarding model training and optimization strategies.

Predictions on the test set are performed using the trained LSTM model and a classification report is generated to evaluate the model's performance.

```
# Make predictions on the test set
y_pred = lstm_model.predict_classes(X_test_reshaped)

# Convert one-hot encoded labels to integers
y_test_classes = np.argmax(y_test_one_hot, axis=1)

# Generate classification report
classification_rep = classification_report(y_test_classes, y_pred)

# Print the classification report
print("Classification Report:")
print(classification_rep)
```

First, the predict_classes method of the LSTM model is used to obtain the predicted class labels for the test set (X_test_reshaped). These predicted labels are stored in the y_pred variable.

Next, the original labels in the test set are converted from one-hot encoded format to integers using the argmax function along the appropriate axis (axis=1). These converted labels are stored in the y_test_classes variable.

Then, the classification_report function from scikit-learn's metrics module is called. This function takes the true labels (y_test_classes) and the predicted labels (y_pred) as inputs and generates a comprehensive report containing various classification metrics.

The classification report provides metrics such as precision, recall, F1-score, and support for each class in the classification task. These metrics help in evaluating the model's performance across different classes and provide insights into its strengths and weaknesses.

Precision, recall, F1-score, and support are used metrics in classification tasks to evaluate the performance of the model.

Precision: Precision measures the proportion of true positive predictions among all positive predictions made by the model. It is calculated as the ratio of true positives to the sum of true positives and false positives. Precision reflects the model's ability to correctly identify positive instances without misclassifying negative instances as positives. A higher precision value indicates fewer false positives.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall (Sensitivity): Recall measures the proportion of true positive predictions among all actual positive instances in the dataset. It is calculated as the ratio of true positives to the sum of true positives and false negatives. Recall reflects the model's ability to capture all positive instances without missing any. A higher recall value indicates fewer false negatives.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

F1-score: The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall, considering both false positives and false negatives. F1-score is calculated as the weighted average of precision and recall, where a higher F1-score indicates better overall performance. It is particularly useful when there is an imbalance between the number of positive and negative instances in the dataset.

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Support: Support represents the number of actual occurrences of each class in the dataset. It is the number of samples in the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) categories. Support values provide context for precision, recall, and F1-score, helping to interpret the significance of these metrics for each class.

```
#Saving model to disk
model_path = "lstm_model.h5"
lstm_model.save(model_path)

print("Model saved at /kaggle/working/")
```

5.1.11 LSTM Loading

```
# Load the LSTM model from disk
lstm_model = tf.keras.models.load_model('/kaggle/working/lstm_model.h5')

# Display model summary
print(lstm_model.summary())
```

The saved model is loaded from the file 'lstm_model.h5' into memory using the load_model() function from the Keras module. Once the model is loaded, it utilises the summary() method to display a comprehensive overview of the model's architecture and parameters. This summary serves as a valuable reference, providing detailed insights into the structure of the LSTM model.

The summary includes essential information such as the total number of layers in the model, the type of each layer (e.g., LSTM, Dense), the number of trainable parameters, and the output shape of each layer. Understanding the architecture and parameters of the model is crucial for various tasks, including model interpretation, optimization, and troubleshooting. By

examining the summary, users can gain a deeper understanding of how information flows through the model and how different layers contribute to the overall computation.

Furthermore, the summary helps in verifying the correctness of the model's architecture, ensuring that it aligns with the intended design. It also facilitates comparison with other models or variations of the same model, enabling researchers and practitioners to make informed decisions about model selection and optimization strategies. Overall, the summary provides valuable insights into the inner workings of the LSTM model.

5.1.12 Testing an Unknown Sample

```
data, sampling_rate = librosa.load('/kaggle/input/sample-data/download.wav')
plt.figure(figsize=(15, 5))
librosa.display.waveplot(data, sr=sampling_rate)
ipd.Audio('/kaggle/input/sample-data/download.wav')
```

```
X, sample_rate = librosa.load('/kaggle/input/sample-data/download.wav'
                              ,res_type='kaiser_fast'
                              ,duration=2.5
                              ,sr=44100
                              ,offset=0.5
                             )

sample_rate = np.array(sample_rate)
mfccs = np.mean(librosa.feature.mfcc(y=X, sr=sample_rate, n_mfcc=20),axis=1)
newdf = pd.DataFrame(data=mfccs).T
newdf
```

The initiation of the emotional inference process by loading an unknown audio sample and visualising its waveform to grasp its fundamental attributes is done. Following this, we play back the audio using IPython.display. Subsequently, it extracts features from the audio data using the MFCC method, consistent with the previous extraction process. These MFCC features are organised into a DataFrame named ‘newdf’, wherein each row encapsulates a frame of MFCC coefficients, and each column denotes a distinct MFCC feature. This structured representation facilitates further analysis and processing of the audio features, enabling us to proceed with the emotional inference task. By systematically extracting and organising the MFCC features, it prepares the audio data for subsequent processing steps, ultimately leading to the identification and interpretation of its emotional content.

```
X_scaled = scaler.transform(newdf)
X_reshaped = X_scaled.reshape(X_scaled.shape[0], 1, X_scaled.shape[1])
X_reshaped
```

Now, feature scaling and reshaping is done on the extracted MFCC features from the unknown audio sample. Feature scaling is a preprocessing step commonly applied in machine learning to ensure that all input features have a similar scale, which can improve the performance and convergence of many machine learning algorithms. Here, the scalar object,

which was trained on the training data, is used to transform the extracted MFCC features to have zero mean and unit variance.

After scaling the features, the X_scaled array contains the scaled MFCC features. The next step involves reshaping the scaled features into a format suitable for input into the LSTM model. LSTM networks expect input data in a three-dimensional format: (batch_size, timesteps, features).

In this snippet, X_reshaped is obtained by reshaping X_scaled such that:

- The first dimension corresponds to the number of frames or samples in the audio sequence.
- The second dimension is set to 1, representing a single time step in the sequence.
- The third dimension corresponds to the number of MFCC features extracted from each frame.

This reshaping operation readies the MFCC features for input into the LSTM model, facilitating effective processing of the temporal sequence of audio features. By reshaping the data in this manner, it ensures compatibility with the LSTM model's input requirements, allowing it to seamlessly handle the sequential nature of the audio features. This preparatory step optimises the model's ability to capture temporal dependencies and nuances within the audio data, enhancing its performance in accurately classifying emotions. Through this process, the LSTM model can effectively leverage the temporal dynamics inherent in the audio features, enabling robust and insightful emotion classification.

```python
# Predict emotion class
predicted_class = lstm_model.predict_classes(X_reshaped)

predicted_labels = []
for index in predicted_class:
    if index == 0:
        predicted_labels.append("female_happy")
    elif index == 1:
        predicted_labels.append("female_fear")
    elif index == 2:
        predicted_labels.append("female_sad")
    elif index == 3:
        predicted_labels.append("female_disgust")
    elif index == 4:
        predicted_labels.append("female_angry")
    elif index == 5:
        predicted_labels.append("female_neutral")
    elif index == 6:
        predicted_labels.append("male_neutral")
    elif index == 7:
        predicted_labels.append("male_sad")
    elif index == 8:
        predicted_labels.append("male_disgust")
    elif index == 9:
        predicted_labels.append("male_fear")
    elif index == 10:
        predicted_labels.append("male_happy")
    elif index == 11:
        predicted_labels.append("male_angry")
    elif index == 12:
        predicted_labels.append("female_surprise")
    elif index == 13:
        predicted_labels.append("male_surprise")

print("Predicted Emotion Class: ")
print(predicted_labels)
```

The LSTM model predicts the emotion class label for the unknown audio sample based on the extracted MFCC features. The predict_classes method of the LSTM model is used to obtain the predicted class indices, which are then mapped to their corresponding emotion labels. Finally the predicted emotion of the unknown audio sample is given as the output.

5.2 Complete Source Code

#IMPORTING LIBRARIES

```
import librosa                  #used for audio processing tasks
import librosa.display          #provides functions for visualising audio data
import numpy as np              #used for numerical computations
import matplotlib.pyplot as plt   #used for creating static and interactive visualisations
from matplotlib.pyplot import specgram
import pandas as pd             #package for data manipulation and library analysis
import glob                     #used for file operations and pathname pattern matching
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import learning_curve
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler     #used for normalisation of data
```

```
from sklearn.neighbors import KNeighborsClassifier   #required for KNN model
from sklearn.svm import SVC                          #required for SVM model
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from sklearn.externals import joblib    #used for model serialisation
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam
import IPython.display as ipd       #used for displaying audio signals
from tqdm import tqdm               #progress bar
import seaborn as sns               #used for visualisation
import pickle                       #used for serialisation of Python objects
import os       #used for file operations and manipulating environment variables
import sys

import warnings
# ignore warnings
if not sys.warnoptions:
    warnings.simplefilter("ignore")
warnings.filterwarnings("ignore", category=DeprecationWarning)

#LOADING DATASETS

# Traversing the directory tree rooted at the input directory
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        os.path.join(dirname, filename)

# Defining the paths to the datasets
TESS = "/kaggle/input/toronto-emotional-speech-set-tess/tess toronto emotional
speech set data/TESS Toronto emotional speech set data/"
RAV = "/kaggle/input/ravdess-emotional-speech-audio/audio_speech_actors_01-24/"
SAVEE = "/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/"
CREMA = "/kaggle/input/cremad/AudioWAV/"

#EXPLORING THE DATASETS

dir_list = os.listdir(SAVEE)
# List initialization to store emotions and file paths
emotion =[]
path = []
```

```
for i in dir_list:
    if i[-8:-6]=='_a':
        emotion.append('male_angry')
    elif i[-8:-6]=='_d':
        emotion.append('male_disgust')
    elif i[-8:-6]=='_f':
        emotion.append('male_fear')
    elif i[-8:-6]=='_h':
        emotion.append('male_happy')
    elif i[-8:-6]=='_n':
        emotion.append('male_neutral')
    elif i[-8:-6]=='sa':
        emotion.append('male_sad')
    elif i[-8:-6]=='su':
        emotion.append('male_surprise')
    else:
        emotion.append('male_error')
    path.append(SAVEE + i)

# Dataframe creation to store emotion labels, file paths and source dataset
SAVEE_df = pd.DataFrame(emotion, columns = ['labels'])
SAVEE_df['source'] = 'SAVEE'
SAVEE_df =pd.concat([SAVEE_df, pd.DataFrame(path, columns = ['path'])], axis=1)
print(SAVEE_df.labels.value_counts())

dir_list = os.listdir(RAV)
dir_list.sort()

# List initialization to store emotions and file paths
emotion = []
gender = []
path = []
for i in dir_list:
    fname = os.listdir(RAV + i)
    for f in fname:
        part = f.split('.')[0].split('-')
        emotion.append(int(part[2]))
        temp = int(part[6])
        if temp%2 == 0:
            temp = "female"
        else:
            temp = "male"
        gender.append(temp)
        path.append(RAV + i + '/' + f)
```

```
# Dataframe creation to store emotion labels, file paths and source dataset
RAV_df = pd.DataFrame(emotion)
RAV_df = RAV_df.replace({1:'neutral', 2:'neutral', 3:'happy', 4:'sad',
                         5:'angry', 6:'fear', 7:'disgust', 8:'surprise'})
RAV_df = pd.concat([pd.DataFrame(gender),RAV_df],axis=1)
RAV_df.columns = ['gender','emotion']
RAV_df['labels'] =RAV_df.gender + '_' + RAV_df.emotion
RAV_df['source'] = 'RAVDESS'
RAV_df = pd.concat([RAV_df,pd.DataFrame(path, columns = ['path'])],axis=1)
RAV_df = RAV_df.drop(['gender', 'emotion'], axis=1)
print(RAV_df.labels.value_counts())

dir_list = os.listdir(TESS)
dir_list.sort()

# List initialization to store emotions and file paths
path = []
emotion = []

for i in dir_list:
    fname = os.listdir(TESS + i)
    for f in fname:
        if i == 'OAF_angry' or i == 'YAF_angry':
            emotion.append('female_angry')
        elif i == 'OAF_disgust' or i == 'YAF_disgust':
            emotion.append('female_disgust')
        elif i == 'OAF_Fear' or i == 'YAF_fear':
            emotion.append('female_fear')
        elif i == 'OAF_happy' or i == 'YAF_happy':
            emotion.append('female_happy')
        elif i == 'OAF_neutral' or i == 'YAF_neutral':
            emotion.append('female_neutral')
        elif i == 'OAF_Pleasant_surprise' or i == 'YAF_pleasant_surprised':
            emotion.append('female_surprise')
        elif i == 'OAF_Sad' or i == 'YAF_sad':
            emotion.append('female_sad')
        else:
            emotion.append('Unknown')
        path.append(TESS + i + "/" + f)

# Dataframe creation to store emotion labels, file paths and source dataset
TESS_df = pd.DataFrame(emotion, columns = ['labels'])
TESS_df['source'] = 'TESS'
```

```
TESS_df = pd.concat([TESS_df,pd.DataFrame(path, columns = ['path'])],axis=1)
print(TESS_df.labels.value_counts())

dir_list = os.listdir(CREMA)
dir_list.sort()

# List initialization to store emotions and file paths
gender = []
emotion = []
path = []
female =
[1002,1003,1004,1006,1007,1008,1009,1010,1012,1013,1018,1020,1021,1024,1025
, 1028, 1029,1030,1037,1043,1046,1047,1049,1052,
1053,1054,1055,1056,1058,1060,1061,1063,
1072,1073,1074,1075,1076,1078,1079,1082,
1084,1089,1091]

for i in dir_list:
    part = i.split('_')
    if int(part[0]) in female:
        temp = 'female'
    else:
        temp = 'male'
    gender.append(temp)
    if part[2] == 'SAD' and temp == 'male':
        emotion.append('male_sad')
    elif part[2] == 'ANG' and temp == 'male':
        emotion.append('male_angry')
    elif part[2] == 'DIS' and temp == 'male':
        emotion.append('male_disgust')
    elif part[2] == 'FEA' and temp == 'male':
        emotion.append('male_fear')
    elif part[2] == 'HAP' and temp == 'male':
        emotion.append('male_happy')
    elif part[2] == 'NEU' and temp == 'male':
        emotion.append('male_neutral')
    elif part[2] == 'SAD' and temp == 'female':
        emotion.append('female_sad')
    elif part[2] == 'ANG' and temp == 'female':
        emotion.append('female_angry')
    elif part[2] == 'DIS' and temp == 'female':
        emotion.append('female_disgust')
    elif part[2] == 'FEA' and temp == 'female':
        emotion.append('female_fear')
```

```
    elif part[2] == 'HAP' and temp == 'female':
        emotion.append('female_happy')
    elif part[2] == 'NEU' and temp == 'female':
        emotion.append('female_neutral')
    else:
        emotion.append('Unknown')
    path.append(CREMA + i)


#Dataframe creation to store emotion labels, file paths and source dataset
CREMA_df = pd.DataFrame(emotion, columns = ['labels'])
CREMA_df['source'] = 'CREMA'
CREMA_df = pd.concat([CREMA_df,pd.DataFrame(path, columns = ['path'])],axis=1)
print(CREMA_df.labels.value_counts())

#COMBINING THE DATASETS

# Concatenate data-frames from different emotion datasets into a single dataframe
df = pd.concat([SAVEE_df, RAV_df, TESS_df, CREMA_df], axis = 0)

# Count the occurrences of each audio label
audio_label_count = df.labels.value_counts()
print(audio_label_count)
print(df.to_csv("Data_path.csv",index=False)) # Saving the combined dataframe to a
CSV file


# Generating bar plot
colors = ['blue', 'green', 'red', 'purple', 'orange',
          'yellow', 'pink', 'brown', 'gray', 'cyan',
          'magenta', 'olive', 'teal', 'navy']
pastel_colors = sns.color_palette("muted", len(colors))

plt.figure(figsize=(22, 10))
plt.bar(audio_label_count.index, audio_label_count.values, color=pastel_colors)
plt.xlabel('Label')
plt.ylabel('Count')
plt.title('Audio Label Count')
plt.xticks(audio_label_count.index)
plt.show()
```

```
# DATA PROCESSING

ref = pd.read_csv("/kaggle/working/Data_path.csv")
num_mfcc = 20  # Number of MFCC coefficients
num_files = 12162  # Number of audio files

# Dataframe initialization
columns = ['mfcc_' + str(i+1) for i in range(num_mfcc)]
df = pd.DataFrame(columns=columns, index=range(num_files))
pbar = tqdm(total=num_files) # Progress bar

# Loop through audio files and extract MFCC features
for idx, path in enumerate(ref.path):
    #Loading the audio file from the specified path and extracting MFCC features
    X, sample_rate = librosa.load(path, res_type='kaiser_fast', duration=2.5,
sr=44100, offset=0.5)
    mfccs = librosa.feature.mfcc(y=X, sr=sample_rate, n_mfcc=num_mfcc)

    mfccs_mean = np.mean(mfccs, axis=1)  # Compute mean of MFCC coefficients
along each column
    df.loc[idx] = mfccs_mean # Storing the mean coefficients in the dataframe
    pbar.update(1)

pbar.close()

print(df.shape)
#Concatenating the two data-frames to match the mfcc features
# of a particular audio file to its emotion label

df_concat = pd.concat([ref, df], axis=1)
# Filling any missing values(NaN) in the dataframe with zero.
# This is done for easier handling of missing data.
df_concat=df_concat.fillna(0)
df_concat.to_csv('features.csv', index=False)

#SPLITTING AND NORMALISING THE DATASET

extracted_features = pd.read_csv("/kaggle/working/features.csv")

X_train, X_test, y_train, y_test =
train_test_split(extracted_features.drop(['path','labels','source'],
                                         axis=1)
                                 , df_concat.labels
                                 , test_size=0.20
```

```
                                , shuffle=True
                                , random_state=42 # Random seed - reproducibility
                               )


# Normalisation of training and testing data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

## MODEL CREATION AND ACCURACY MEASUREMENT

#KNN MODEL

knn = KNeighborsClassifier(n_neighbors=5, weights='uniform', metric='euclidean')
# Training
knn.fit(X_train_scaled, y_train)
# Predict labels
y_pred = knn.predict(X_test_scaled)
# Evaluating the classifier and calculating accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
# Saving the trained model to disk
joblib.dump(knn, 'knn_model.pkl')
print("Model saved at: /kaggle/working/")

#SVM MODEL

X_train_array = np.array(X_train)
X_test_array = np.array(X_test)

# Reshape the array if necessary
X_train_flattened = X_train_array.reshape(X_train_array.shape[0], -1)
X_test_flattened = X_test_array.reshape(X_test_array.shape[0], -1)
# Initialize SVM classifier
svm_classifier = SVC(kernel='linear')
# Train the SVM classifier
svm_classifier.fit(X_train_flattened, y_train)
# Predict labels for test data
y_pred_svm = svm_classifier.predict(X_test_flattened)
# Calculate accuracy
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print("Accuracy:", accuracy_svm)
joblib.dump(svm_classifier, 'svm_model.pkl')
```

```
print("Model saved at /kaggle/working/")

#LSTM MODEL

# Reshape input data for LSTM
X_train_reshaped = X_train_scaled.reshape(X_train_scaled.shape[0], 1,
X_train_scaled.shape[1])
X_test_reshaped = X_test_scaled.reshape(X_test_scaled.shape[0], 1,
X_test_scaled.shape[1])

# One-hot encode the labels
label_binarizer = LabelBinarizer()
y_train_one_hot = label_binarizer.fit_transform(y_train)
y_test_one_hot = label_binarizer.transform(y_test)
num_classes = 14

# Defining the LSTM model
lstm_model = Sequential([
   LSTM(units=128, input_shape=(X_train_reshaped.shape[1],
X_train_reshaped.shape[2])),
   Dropout(0.5),
   Dense(64, activation='relu'),
   Dense(num_classes, activation='softmax')  # Use 'softmax' activation for multi-
class classification
])

# Compiling the model
lstm_model.compile(optimizer=Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])

# Training the model
history = lstm_model.fit(X_train_reshaped, y_train_one_hot, epochs=100,
batch_size=16, validation_data=(X_test_reshaped, y_test_one_hot))

# Evaluating the model and calculate accuracy
loss, accuracy = lstm_model.evaluate(X_test_reshaped, y_test_one_hot)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)

#GENERATING CLASSIFICATION REPORT FOR LSTM

# Make predictions on the test set
y_pred = lstm_model.predict_classes(X_test_reshaped)
# Convert one-hot encoded labels to integers
```

```
y_test_classes = np.argmax(y_test_one_hot, axis=1)
# Generate classification report
classification_rep = classification_report(y_test_classes, y_pred)
# Print the classification report
print("Classification Report:")
print(classification_rep)
#Saving model to disk
model_path = "lstm_model.h5"
lstm_model.save(model_path)
print("Model saved at /kaggle/working/")

#TESTING UNKNOWN SAMPLE ON LSTM MODEL

# Load the LSTM model from disk
lstm_model = tf.keras.models.load_model('/kaggle/working/lstm_model.h5')
# Display model summary
print(lstm_model.summary())

data, sampling_rate = librosa.load('/kaggle/input/sample-data/download.wav')
plt.figure(figsize=(15, 5))
librosa.display.waveplot(data, sr=sampling_rate)
ipd.Audio('/kaggle/input/sample-data/download.wav')

X, sample_rate = librosa.load('/kaggle/input/sample-data/download.wav'
                              ,res_type='kaiser_fast'
                              ,duration=2.5
                              ,sr=44100
                              ,offset=0.5
                             )

sample_rate = np.array(sample_rate)
mfccs = np.mean(librosa.feature.mfcc(y=X, sr=sample_rate, n_mfcc=20),axis=1)
newdf = pd.DataFrame(data=mfccs).T
newdf

X_scaled = scaler.transform(newdf)
X_reshaped = X_scaled.reshape(X_scaled.shape[0], 1, X_scaled.shape[1])
X_reshaped

# Predict emotion class
predicted_class = lstm_model.predict_classes(X_reshaped)

predicted_labels = []
for index in predicted_class:
```

```
    if index == 0:
        predicted_labels.append("female_happy")
    elif index == 1:
        predicted_labels.append("female_fear")
    elif index == 2:
        predicted_labels.append("female_sad")
    elif index == 3:
        predicted_labels.append("female_disgust")
    elif index == 4:
        predicted_labels.append("female_angry")
    elif index == 5:
        predicted_labels.append("female_neutral")
    elif index == 6:
        predicted_labels.append("male_neutral")
    elif index == 7:
        predicted_labels.append("male_sad")
    elif index == 8:
        predicted_labels.append("male_disgust")
    elif index == 9:
        predicted_labels.append("male_fear")
    elif index == 10:
        predicted_labels.append("male_happy")
    elif index == 11:
        predicted_labels.append("male_angry")
    elif index == 12:
        predicted_labels.append("female_surprise")
    elif index == 13:
        predicted_labels.append("male_surprise")

print("Predicted Emotion Class: ")
print(predicted_labels)
```

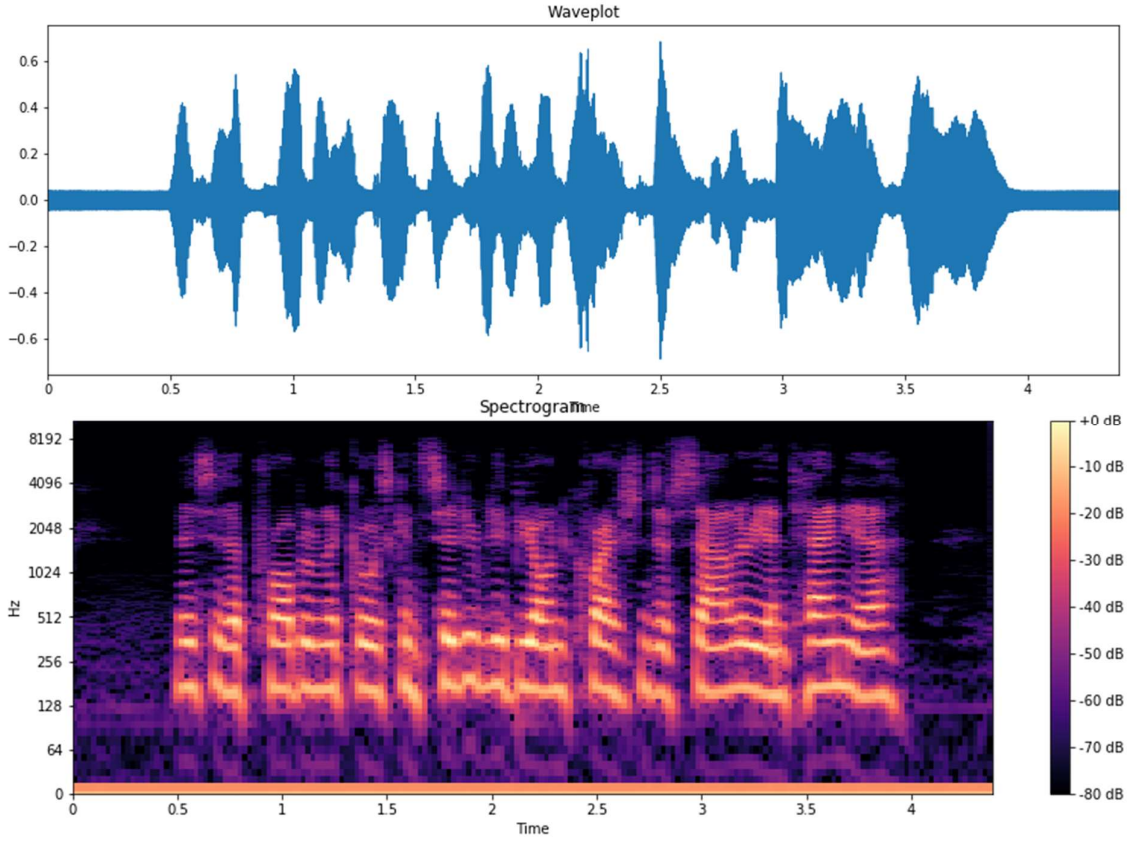
5.3 Unit Testing

Testing an audio sample of SAVEE dataset:

```
#Testing an audio sample - SAVEE
fname = SAVEE + 'DC_f12.wav'
data, sampling_rate = librosa.load(fname)
plt.figure(figsize=(15, 5))
librosa.display.waveplot(data, sr=sampling_rate)
plt.title('Waveplot') # Displaying Waveplot

plt.figure(figsize=(15, 5))
D = librosa.amplitude_to_db(librosa.stft(data), ref=np.max)
librosa.display.specshow(D, sr=sampling_rate, x_axis='time', y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram') # Displaying Spectogram
plt.show()

ipd.Audio(fname) # Audio Playback
```

Output:

Given below is the waveplot and spectrogram of an arbitrary audio sample from the dataset.

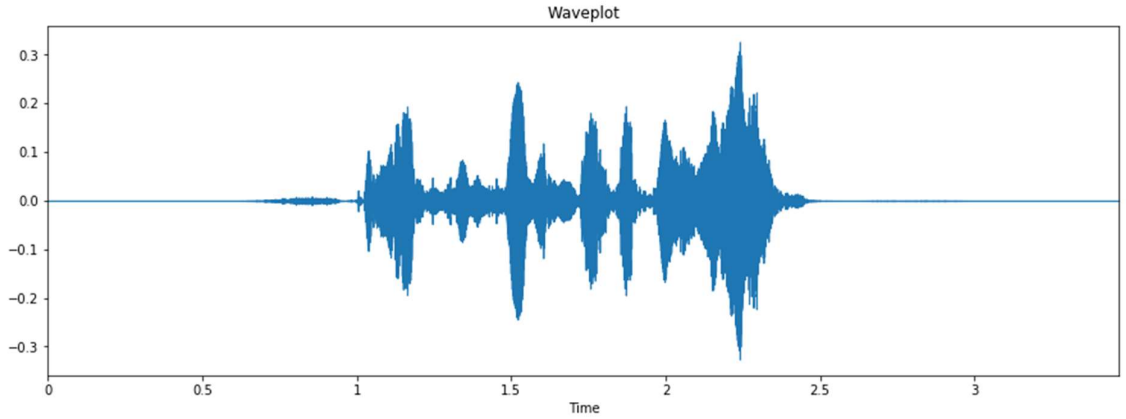


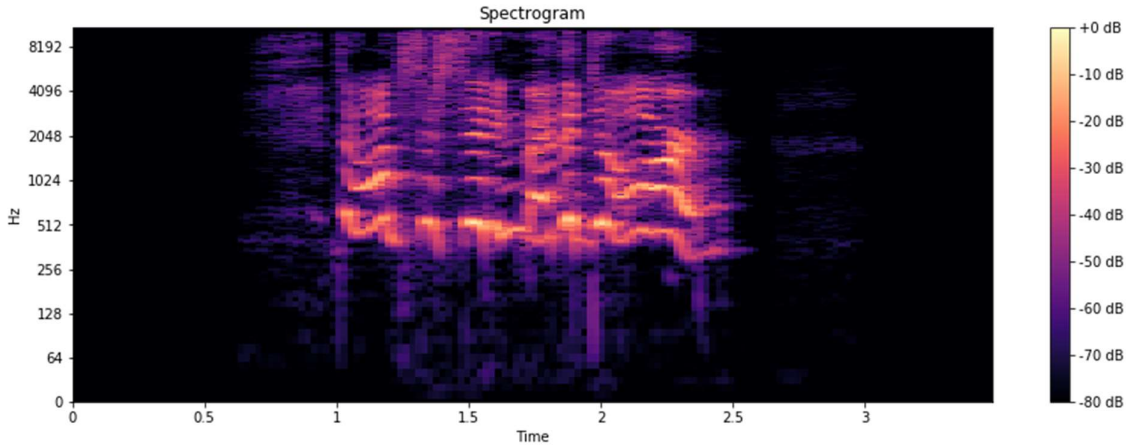
Testing an audio sample of RAVDESS dataset:

```
#Testing an audio sample - RAVDESS
fname = RAV + 'Actor_14/03-01-06-02-02-02-14.wav'
data, sampling_rate = librosa.load(fname)
plt.figure(figsize=(15, 5))
librosa.display.waveplot(data, sr=sampling_rate)
plt.title('Waveplot') # Displaying Waveplot

plt.figure(figsize=(15, 5))
D = librosa.amplitude_to_db(librosa.stft(data), ref=np.max)
librosa.display.specshow(D, sr=sampling_rate, x_axis='time', y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram') # Displaying Spectogram
plt.show()
```

Output:

Given below is the waveplot and spectrogram of an arbitrary audio sample from the dataset.



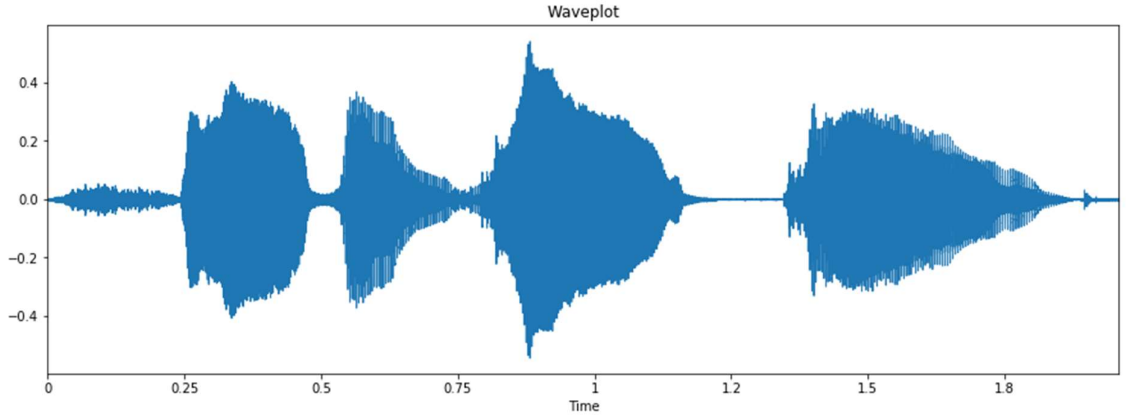


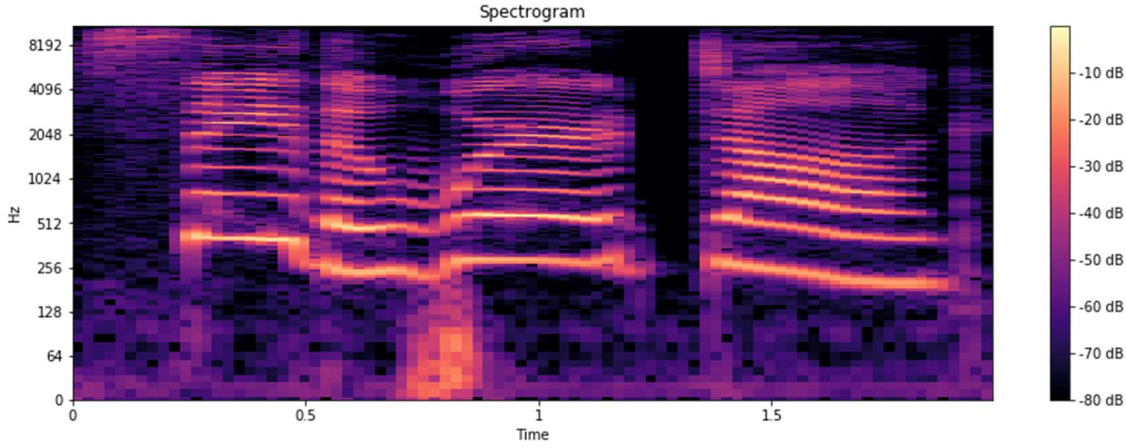
Testing an audio sample of TESS dataset:

```
#Testing an audio sample - TESS
fname = TESS + 'YAF_happy/YAF_dog_happy.wav'

data, sampling_rate = librosa.load(fname)
plt.figure(figsize=(15, 5))
librosa.display.waveplot(data, sr=sampling_rate)
plt.title('Waveplot') # Displaying Waveplot

plt.figure(figsize=(15, 5))
D = librosa.amplitude_to_db(librosa.stft(data), ref=np.max)
librosa.display.specshow(D, sr=sampling_rate, x_axis='time', y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram') # Displaying Spectogram
plt.show()
```

Output:

Given below is the waveplot and spectrogram of an arbitrary audio sample from the dataset.



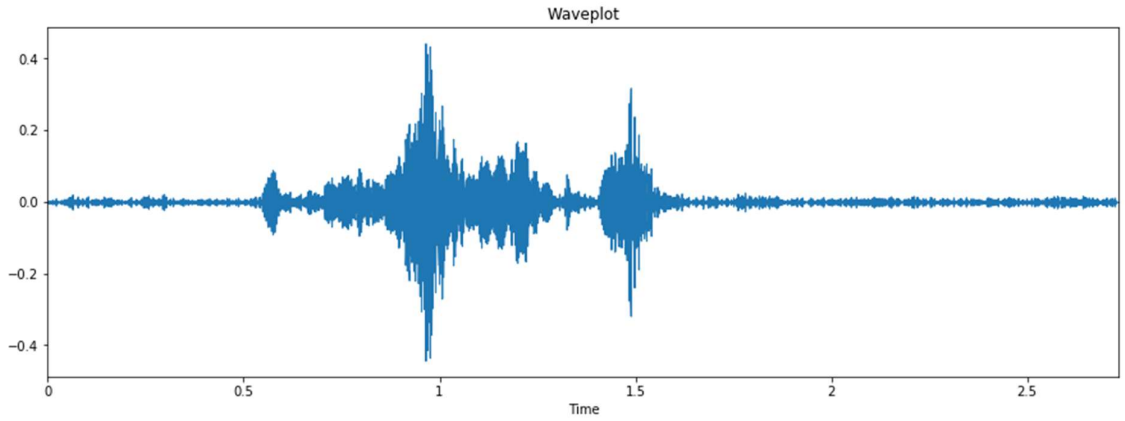


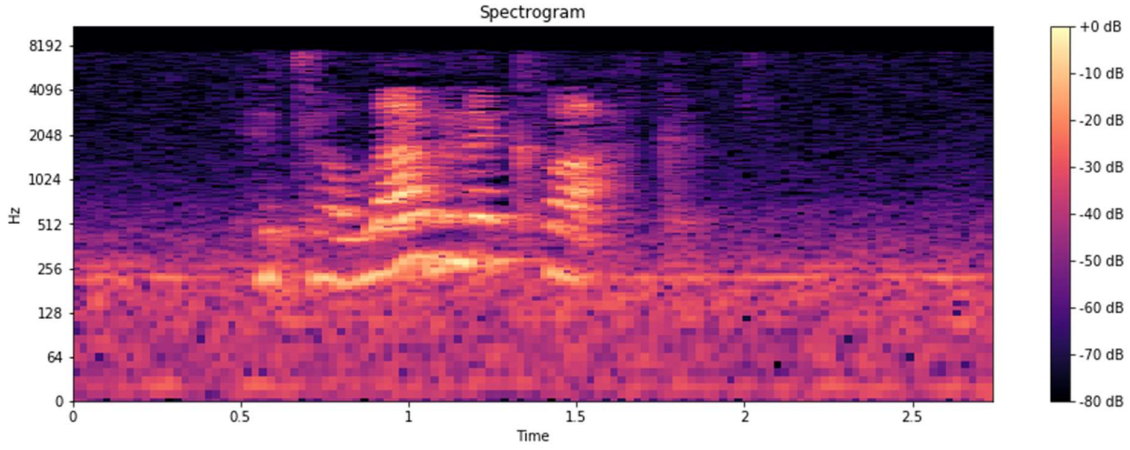
Testing an audio sample of CREMA-D dataset:

```
#Testing an audio sample - CREMA-D
fname = CREMA + '1012_IEO_HAP_HI.wav'
data, sampling_rate = librosa.load(fname)
plt.figure(figsize=(15, 5))
librosa.display.waveplot(data, sr=sampling_rate)
plt.title('Waveplot') # Displaying Waveplot

plt.figure(figsize=(15, 5))
D = librosa.amplitude_to_db(librosa.stft(data), ref=np.max)
librosa.display.specshow(D, sr=sampling_rate, x_axis='time', y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram') # Displaying Spectogram
plt.show()
```

Output:

Given below is the waveplot and spectrogram of an arbitrary audio sample from the dataset.

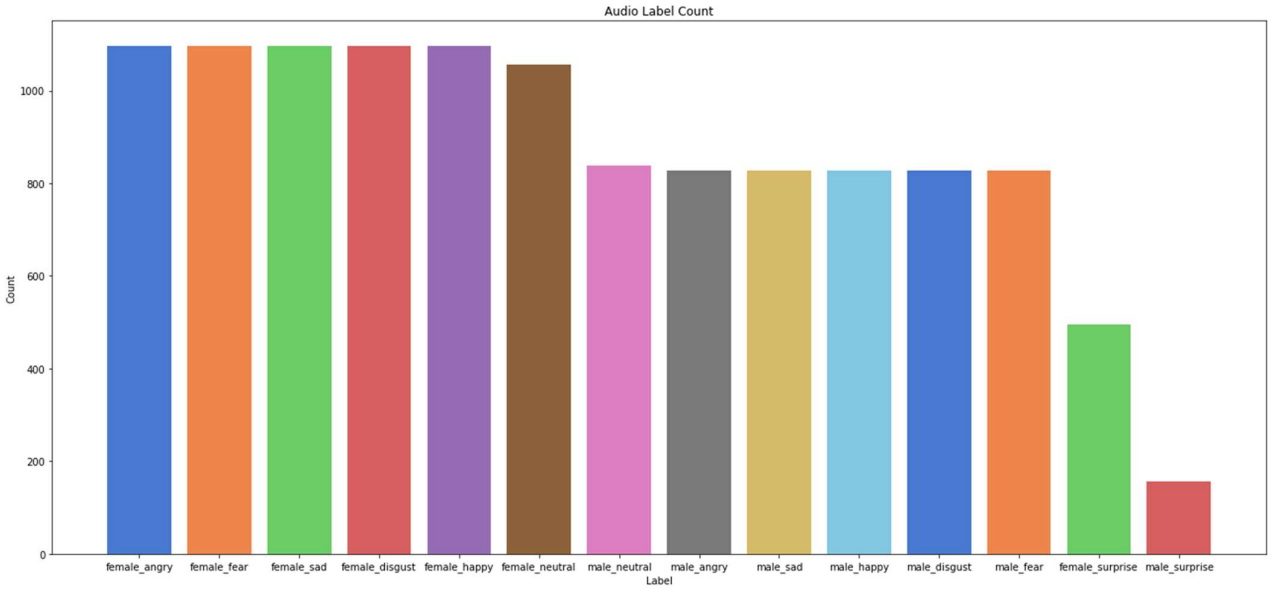




After combining the data from the four datasets, we get the following distribution of the combined dataset which is also stored in a CSV file 'Data_path.csv'. They are listed as follows:

Output:

female_angry 1096
female_fear 1096
female_sad 1096
female_disgust 1096
female_happy 1096
female_neutral 1056
male_neutral 839
male_angry 827
male_sad 827
male_happy 827
male_disgust 827
male_fear 827
female_surprise 496
male_surprise 156
Name: labels, dtype: int64



Contents of first few rows of Data_path.csv:

labels	source	path
male_happy	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_h09.wav
male_fear	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_f12.wav
male_happy	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/DC_h03.wav
male_disgust	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/DC_d04.wav
male_angry	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_a14.wav
male_fear	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_f01.wav
male_happy	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_h15.wav
male_surprise	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JK_su02.wav
male_angry	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JK_a06.wav
male_neutral	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/DC_n01.wav
male_fear	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/DC_f01.wav
male_neutral	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JK_n22.wav
male_angry	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_a10.wav
male_neutral	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_n08.wav
male_sad	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_sa13.wav
male_angry	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/DC_a12.wav
male_fear	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_f07.wav
male_disgust	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_d05.wav
male_disgust	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_d09.wav
male_neutral	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/DC_n30.wav
male_happy	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_h05.wav
male_fear	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_f11.wav
male_disgust	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_d08.wav
male_disgust	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/JE_d04.wav
male_happy	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emotion-savee/ALL/KL_h09.wav

Data_path

```
print(df.shape)
#Concatenating the two dataframes to match the mfcc features
#of a particular audio file to its emotion label
df_concat = pd.concat([ref, df], axis=1)
# Filling any missing values(NaN) in the dataframe with zero.
# This is done for easier handling of missing data.
df_concat=df_concat.fillna(0)
df_concat.to_csv('features.csv', index=False)
print(df_concat.shape)
```

Output:

(12162, 20)
(12162, 23)

Explanation of Output:

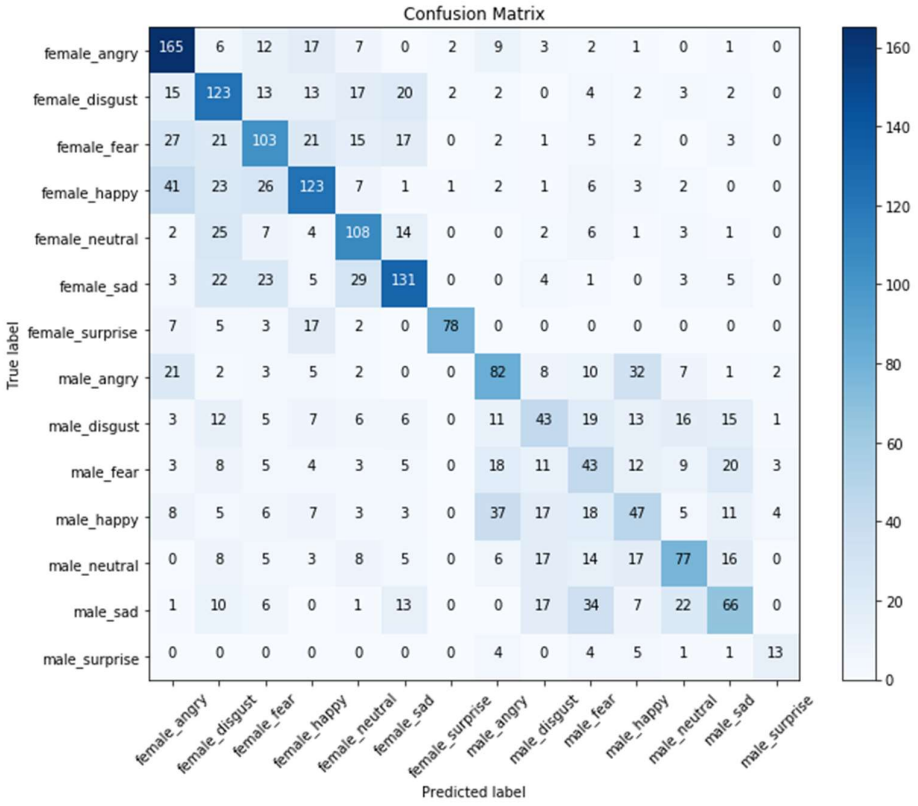
After extracting the 20 MFCC features and storing it in a dataframe, it is appended with the emotion_labels, source and path of the audio files to create a new dataframe. This is stored in a CSV file named "**features.csv".**

Contents of first few rows of features.csv

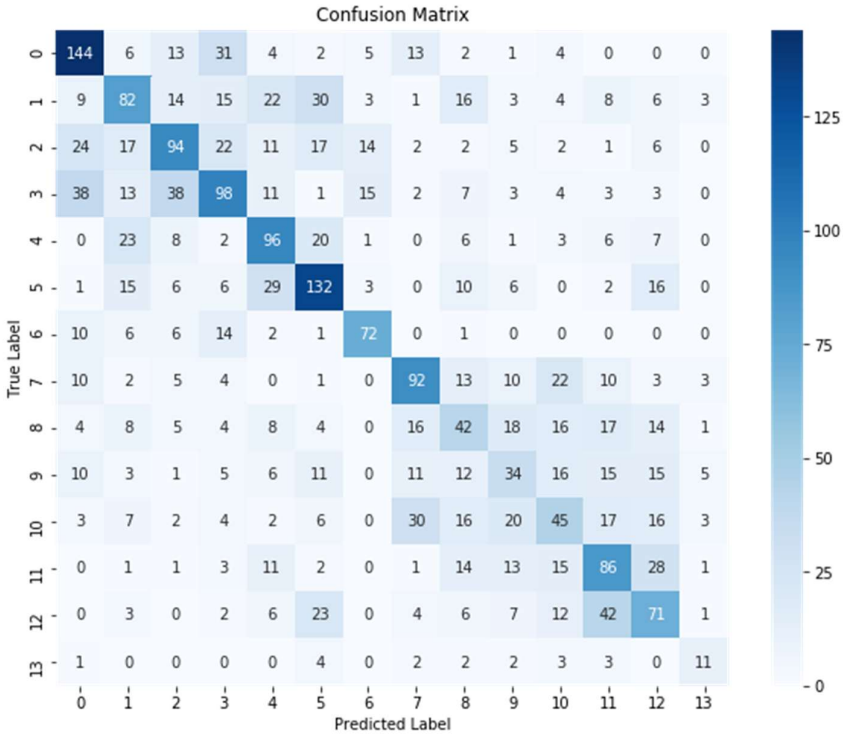
	labels	source	path	mfcc_1	mfcc_2	mfcc_3	mfcc_4	mfcc_5	mfcc_6	mfcc_7	...	mfcc_11	mfcc_12	mfcc_13	mfcc_14	mfcc_15	mfcc_16	mfcc_17	mfcc_18	mfcc_19	mfcc_20
0	male_happy	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emo...	-309.609619	156.562469	12.748092	24.563248	5.987000	-13.370072	-0.902566	...	-16.751169	0.902141	2.987767	-8.257284	-1.060103	-4.434721	-5.881765	4.145817	-2.087474	-8.433018
1	male_fear	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emo...	-621.123352	161.778122	35.570583	36.647305	25.223625	-0.475460	-0.188156	...	-13.794344	-9.369884	-3.399983	-3.791615	-1.427450	1.159548	0.605723	-0.375765	4.168525	4.737456
2	male_happy	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emo...	-344.603577	113.925369	15.898048	13.738302	21.520544	13.897222	-0.368703	...	-6.870913	-9.739623	-0.659857	1.005181	-0.994036	-4.372424	1.068029	4.855748	-4.877567	-3.713228
3	male_disgust	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emo...	-402.221893	133.746902	24.931446	36.427547	42.442593	18.489746	0.040821	...	-6.487161	-17.031794	-1.840788	1.950772	-4.698260	-1.198288	9.812835	8.472905	-2.847432	0.185765
4	male_angry	SAVEE	/kaggle/input/surrey-audiovisual-expressed-emo...	-549.466125	154.088684	12.996944	21.621086	28.807652	4.498013	-2.225336	...	-7.490743	-8.456983	-0.670760	-3.446213	-3.872377	4.194579	3.619041	-2.480716	1.123396	2.933870

5 rows × 23 columns

Confusion Matrix of KNN Model:



Confusion Matrix of SVM Model:



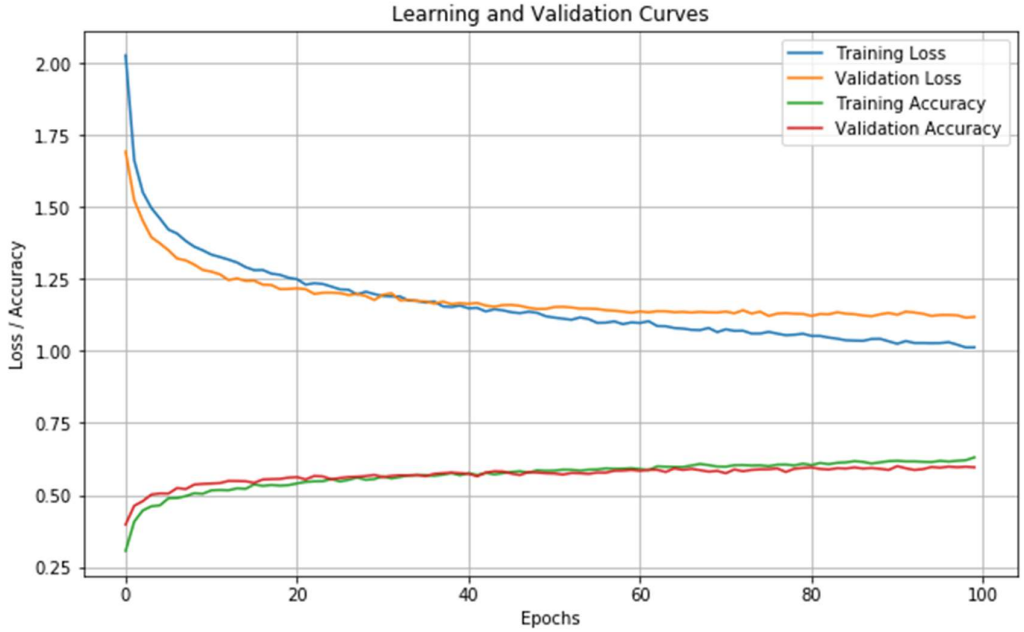
LSTM model training and validation result/**output**:

Train on 9729 samples, validate on 2433 samples
Epoch 1/100
9729/9729 [==============================] - 4s 363us/sample - loss: 2.0286 - acc: 0.3055 - val_loss: 1.6936 - val_acc: 0.3975
Epoch 2/100
9729/9729 [==============================] - 3s 293us/sample - loss: 1.6636 - acc: 0.4072 - val_loss: 1.5246 - val_acc: 0.4624
Epoch 3/100
9729/9729 [==============================] - 3s 298us/sample - loss: 1.5525 - acc: 0.4463 - val_loss: 1.4531 - val_acc: 0.4784
Epoch 4/100
9729/9729 [==============================] - 3s 292us/sample - loss: 1.4975 - acc: 0.4605 - val_loss: 1.3959 - val_acc: 0.5014
Epoch 5/100
9729/9729 [==============================] - 3s 292us/sample - loss: 1.4611 - acc: 0.4639 - val_loss: 1.3747 - val_acc: 0.5051
Epoch 6/100

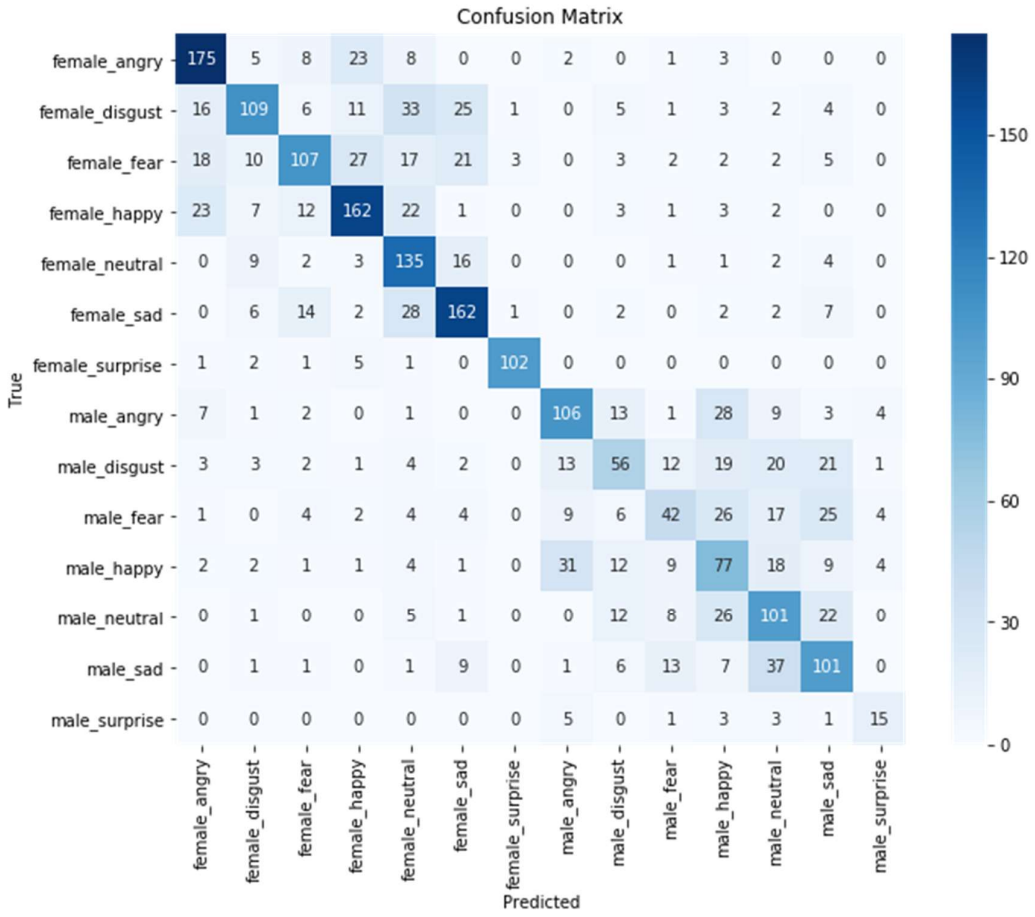
```
9729/9729 [==============================] - 3s 291us/sample - loss: 1.4225 - acc: 0.4885 - val_loss:
1.3506 - val_acc: 0.5047
Epoch 7/100
9729/9729 [==============================] - 3s 314us/sample - loss: 1.4088 - acc: 0.4892 - val_loss:
1.3220 - val_acc: 0.5240
Epoch 8/100
9729/9729 [==============================] - 3s 310us/sample - loss: 1.3833 - acc: 0.4958 - val_loss:
1.3156 - val_acc: 0.5203
Epoch 9/100
9729/9729 [==============================] - 3s 294us/sample - loss: 1.3633 - acc: 0.5059 - val_loss:
1.3009 - val_acc: 0.5360
Epoch 10/100
9729/9729 [==============================] - 3s 298us/sample - loss: 1.3504 - acc: 0.5041 - val_loss:
1.2822 - val_acc: 0.5384
.
.
.
Epoch 35/100
9729/9729 [==============================] - 3s 295us/sample - loss: 1.1745 - acc: 0.5696 - val_loss:
1.1744 - val_acc: 0.5693
Epoch 36/100
9729/9729 [==============================] - 3s 294us/sample - loss: 1.1688 - acc: 0.5691 - val_loss:
1.1727 - val_acc: 0.5656
Epoch 37/100
9729/9729 [==============================] - 3s 297us/sample - loss: 1.1723 - acc: 0.5666 - val_loss:
1.1654 - val_acc: 0.5734
Epoch 38/100
9729/9729 [==============================] - 3s 297us/sample - loss: 1.1557 - acc: 0.5705 - val_loss:
1.1723 - val_acc: 0.5758
Epoch 40/100
9729/9729 [==============================] - 3s 313us/sample - loss: 1.1583 - acc: 0.5683 - val_loss:
1.1668 - val_acc: 0.5758
Epoch 41/100
9729/9729 [==============================] - 3s 301us/sample - loss: 1.1491 - acc: 0.5760 - val_loss:
1.1642 - val_acc: 0.5738
Epoch 42/100
9729/9729 [==============================] - 3s 293us/sample - loss: 1.1505 - acc: 0.5691 - val_loss:
1.1674 - val_acc: 0.5647
Epoch 43/100
9729/9729 [==============================] - 3s 293us/sample - loss: 1.1382 - acc: 0.5767 - val_loss:
1.1584 - val_acc: 0.5775
.
.
.
Epoch 90/100
9729/9729 [==============================] - 3s 300us/sample - loss: 1.0341 - acc: 0.6173 - val_loss:
1.1328 - val_acc: 0.5869
Epoch 91/100
9729/9729 [==============================] - 3s 293us/sample - loss: 1.0256 - acc: 0.6186 - val_loss:
1.1265 - val_acc: 0.6001
Epoch 92/100
```

9729/9729 [==============================] - 3s 293us/sample - loss: 1.0351 - acc: 0.6165 - val_loss: 1.1373 - val_acc: 0.5927
Epoch 93/100
9729/9729 [==============================] - 3s 310us/sample - loss: 1.0283 - acc: 0.6166 - val_loss: 1.1343 - val_acc: 0.5869
Epoch 94/100
9729/9729 [==============================] - 3s 307us/sample - loss: 1.0283 - acc: 0.6152 - val_loss: 1.1300 - val_acc: 0.5898
Epoch 95/100
9729/9729 [==============================] - 3s 301us/sample - loss: 1.0277 - acc: 0.6142 - val_loss: 1.1222 - val_acc: 0.5972
Epoch 96/100
9729/9729 [==============================] - 3s 297us/sample - loss: 1.0281 - acc: 0.6185 - val_loss: 1.1256 - val_acc: 0.5947
Epoch 97/100
9729/9729 [==============================] - 3s 293us/sample - loss: 1.0311 - acc: 0.6159 - val_loss: 1.1259 - val_acc: 0.5984
Epoch 98/100
9729/9729 [==============================] - 3s 292us/sample - loss: 1.0224 - acc: 0.6188 - val_loss: 1.1244 - val_acc: 0.5964
Epoch 99/100
9729/9729 [==============================] - 3s 293us/sample - loss: 1.0128 - acc: 0.6208 - val_loss: 1.1163 - val_acc: 0.5980
Epoch 100/100
9729/9729 [==============================] - 3s 296us/sample - loss: 1.0130 - acc: 0.6305 - val_loss: 1.1190 - val_acc: 0.5960
2433/2433 [==============================] - 0s 67us/sample - loss: 1.1190 - acc: 0.5960
Test Loss: 1.102037496372353
Test Accuracy: 0.60567206

LSTM Learning Curve:

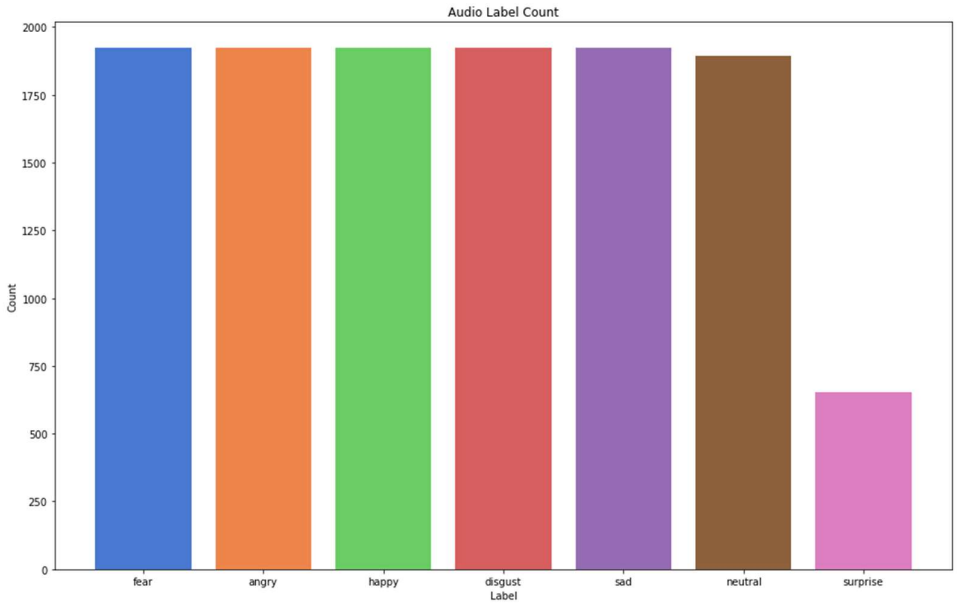


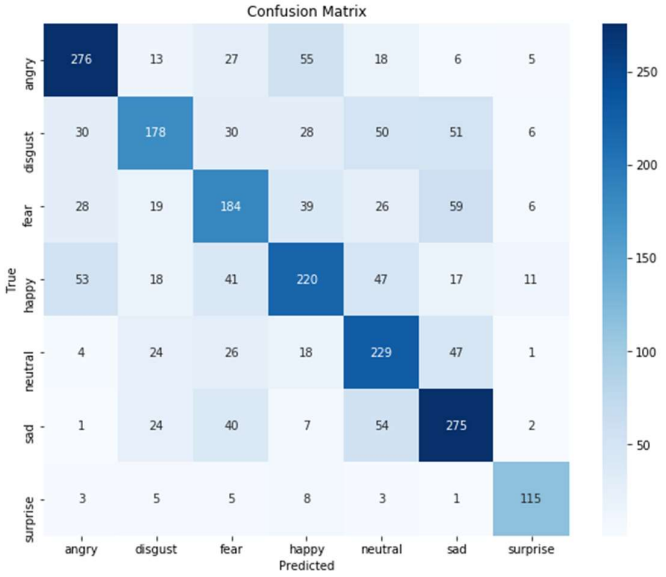
LSTM Confusion Matrix:

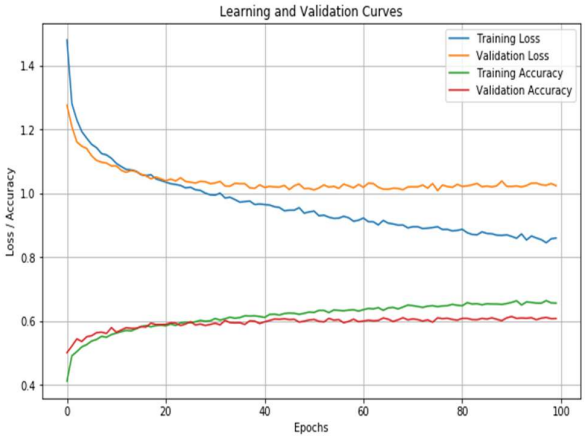


5.4 Unit Testing with 7 classes

Upon observing the LSTM model's accuracy using 14 distinct classes to be approximately 60%, an attempt was made to reduce the number of classes to 7. This entailed consolidating gender-specific classes such as "male_sad" and "female_sad" into a single "sad" class. Subsequently, a new LSTM model was retrained, yielding an incremental increase in accuracy of approximately 2 to 3%. The outputs of this new model, encompassing the emotion distribution among the seven classes via a bar graph, the confusion matrix, learning curve, and the classification report, are depicted in the following images.







Classification Report:

	precision	recall	f1-score	support
0	0.70	0.69	0.69	400
1	0.63	0.48	0.54	373
2	0.52	0.51	0.52	361
3	0.59	0.54	0.56	407
4	0.54	0.66	0.59	349
5	0.60	0.68	0.64	403
6	0.79	0.82	0.80	140
accuracy			0.61	2433
macro avg	0.62	0.63	0.62	2433
weighted avg	0.61	0.61	0.61	2433

5.5 Integrated/Integration Testing

Integration testing is a software testing technique where individual software modules or components are combined and tested as a group. The goal is to verify that these integrated modules work together as expected and interact correctly, ensuring that the software system functions as intended. Integration testing helps identify any issues or defects that may arise from the interaction between different modules, ensuring the overall system's reliability, performance, and functionality.

During integration testing, our module exhibited a notable trend: for certain emotion classes, it consistently predicted the correct emotion, while for others, it occasionally made incorrect predictions. However, these erroneous predictions were often in close proximity to the actual emotion of the audio file. To illustrate, consider the "male_sad" emotion class. While the model rarely predicted emotions like "male_happy" or "female_happy" for these samples, it occasionally misclassified them as "male_disgust". This phenomenon underscores the subjective nature of emotion perception; what one person interprets as sadness, another might perceive as disgust. Despite these occasional misclassifications, the model's predictions remained closely aligned with the actual emotions expressed in the audio files, demonstrating the nuanced nature of emotion recognition and the challenges inherent in capturing subtle emotional distinctions.

Furthermore, our model demonstrated high accuracy in determining the gender of the individual in the audio file. For instance, when classifying samples as "male_happy," the model consistently identified them as belonging to the male gender and almost never misclassified them as "female_happy".

CHAPTER 6: RESULTS AND DISCUSSION

6.1 Analysis of the Dataset

The main dataset is made by combining 4 different datasets, namely SAVEE, RAVDESS, TESS, and CREMA-D. The Combined Dataset used is as follows:

Sl. No.	Emotion Class	Quantity of Audio Samples
1	female_angry	1096
2	female_fear	1096
3	female_sad	1096
4	female_disgust	1096
5	female_happy	1096
6	female_neutral	1056
7	male_neutral	839
8	male_angry	827
9	male_sad	827
10	male_happy	827
11	male_disgust	827
12	male_fear	827
13	female_surprise	496
14	male_surprise	156

Table 6.1.1: Audio Dataset Samples

The dataset provided contains labelled audio samples categorised into various emotion classes, with a total of 14 distinct classes. Each class represents a specific emotional state, including anger, fear, sadness, disgust, happiness, neutral, and surprise, for both male and female genders. The dataset is well-balanced, with each emotion class having a relatively similar number of samples, ranging from 827 to 1096 instances. However, there are notable differences in the number of samples between male and female genders, with some classes having fewer samples for males compared to females.

This dataset is suitable for speech emotion classification tasks as it covers a diverse range of emotions commonly expressed in human speech. The inclusion of multiple emotional states allows for comprehensive training and evaluation of emotion recognition models. Additionally, the balanced distribution of samples across different classes helps prevent biases and ensures that the model learns to distinguish between various emotions effectively.

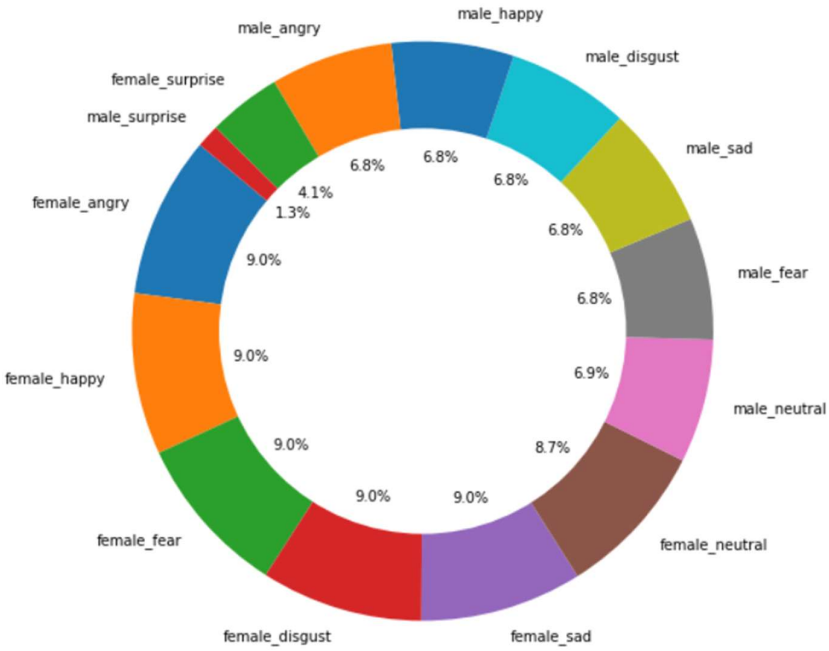


Fig 6.1.1: A Pie Chart Showing the emotion distribution in the Combined Dataset

The dataset's gender-specific labelling enables the development of gender-aware emotion recognition models, which can account for gender-related differences in speech patterns and emotional expression. For example, the model can learn to recognize subtle variations in intonation, pitch, and vocal characteristics that may vary between male and female speakers.

The presence of multiple samples for each emotion class enhances the robustness of the emotion classification model. With a sufficient number of samples per class, the model can learn to generalise well across different instances of the same emotion, reducing the risk of overfitting and improving its performance on unseen data.

Furthermore, the inclusion of the "neutral" emotion class provides a baseline for comparison and allows the model to differentiate between neutral and non-neutral emotional states. This distinction is essential for accurately identifying emotional cues in speech and avoiding misclassification of neutral expressions as other emotions.

Overall, the dataset provides a comprehensive and balanced collection of labelled audio samples for speech emotion classification tasks. Its gender-specific labelling, diverse range of

emotion classes, and sufficient sample size per class make it well-suited for training and evaluating emotion recognition models. Researchers and practitioners can leverage this dataset to develop robust and accurate speech emotion classification systems capable of accurately identifying and interpreting emotional cues in human speech across various contexts and applications.

However, the lack of "male_surprise" class audio files presents a significant challenge for the emotion classification model, potentially affecting its training, generalisation ability, evaluation, and overall performance.

6.2 Comparative Study of the used Models

In this speech emotion recognition project, SVM (Support Vector Machine), KNN (K-Nearest Neighbors), and LSTM (Long Short-Term Memory) were evaluated as potential models. Each of these models has distinct characteristics and advantages, which were compared across various aspects to determine the most suitable model for the task.

Let us first compare the complexities of the models. SVM is a relatively simple model that constructs a hyperplane to separate different classes in feature space. It is based on the principle of finding the maximum margin between classes.

KNN is a non-parametric and instance-based algorithm that classifies instances based on their similarity to neighbouring instances in feature space. It is intuitive and easy to understand.

LSTM is a type of recurrent neural network (RNN) designed to capture temporal dependencies in sequential data. It consists of memory cells and gates to regulate information flow over time, making it more complex than SVM and KNN.

Now let us discuss the capability of the models to handle the temporal dependencies or the sequential data. SVM is not inherently designed to handle sequential or temporal data. It typically requires feature engineering to represent temporal patterns effectively, which limits its performance in speech emotion recognition tasks.

KNN does not explicitly model temporal dependencies and treats each sample independently. While it can capture local patterns, it struggles to capture long-range dependencies present in sequential data like speech.

LSTM is specifically designed for sequential data processing and is well-suited for tasks involving temporal dependencies. It can effectively capture long-term patterns and dynamics in speech signals, making it a suitable choice for speech emotion recognition.

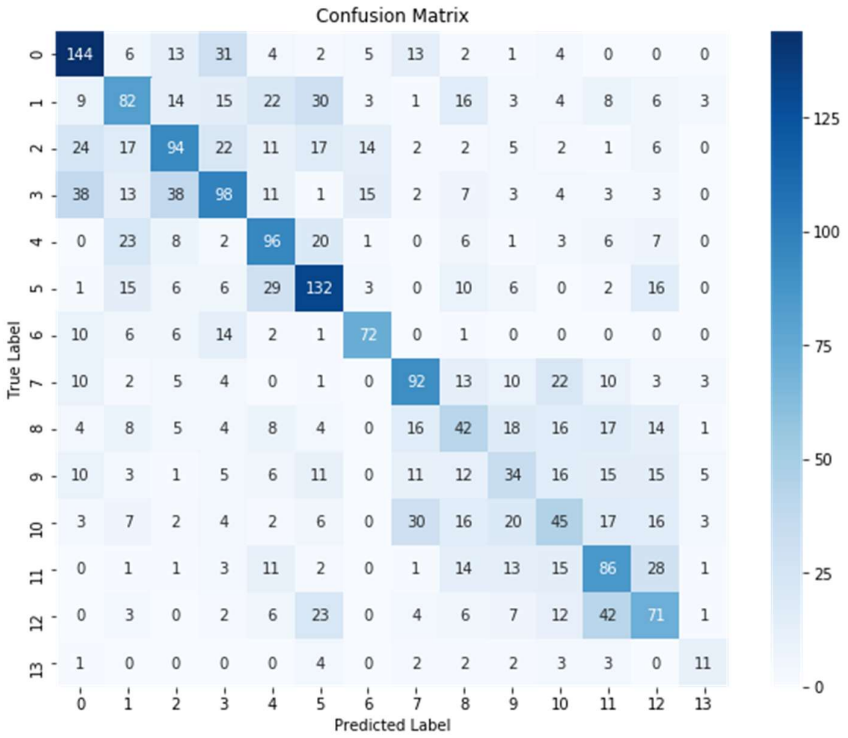
Now let us see how the models do when training time and scalability is concerned. SVM training time was relatively high, especially because the dataset was large, as it involves solving a convex optimization problem. However, it is generally considered scalable and efficient for moderate-sized datasets.

KNN does not involve explicit training and stores the entire training dataset in memory, making it memory-intensive and slow to query, especially for the large dataset with high-dimensional feature space.

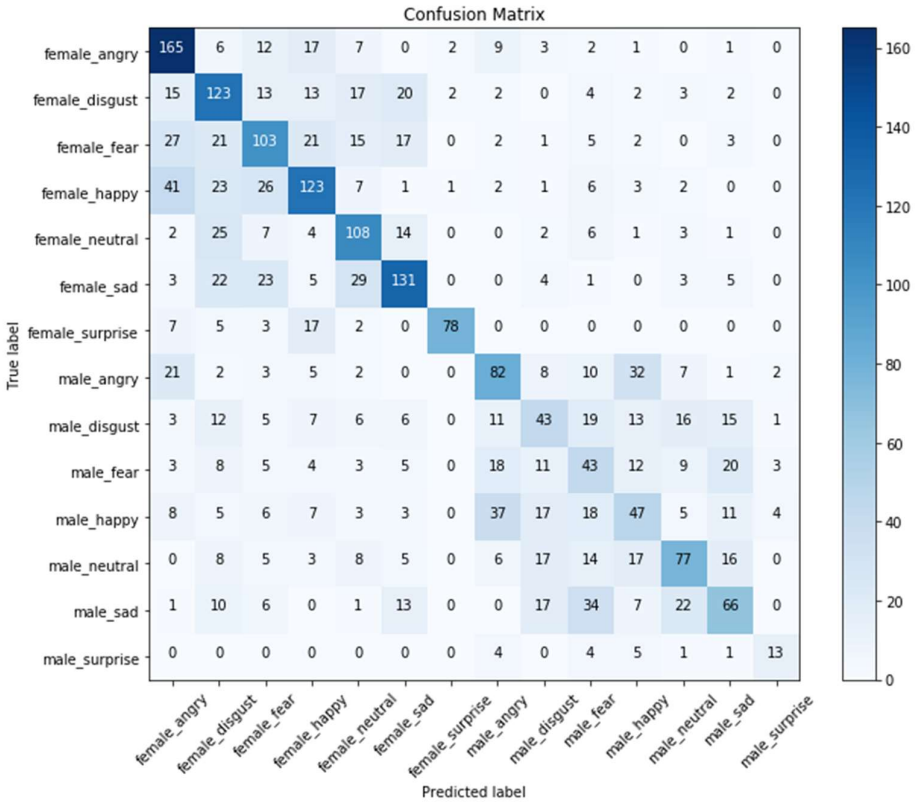
LSTM training time can be significant, especially for deep and complex architectures, as it involves iterative optimization through backpropagation through time (BPTT). However, it offers scalability with the availability of parallel computing resources and can handle large datasets efficiently.

Based on these comparative aspects, LSTM emerges as the preferred model for speech emotion recognition tasks due to its ability to handle temporal data effectively, capture long-term dependencies, and offer scalability with parallel computing resources. While SVM and KNN have their strengths in interpretability and simplicity, respectively, they may not be as well-suited for tasks involving sequential data like speech emotion recognition.

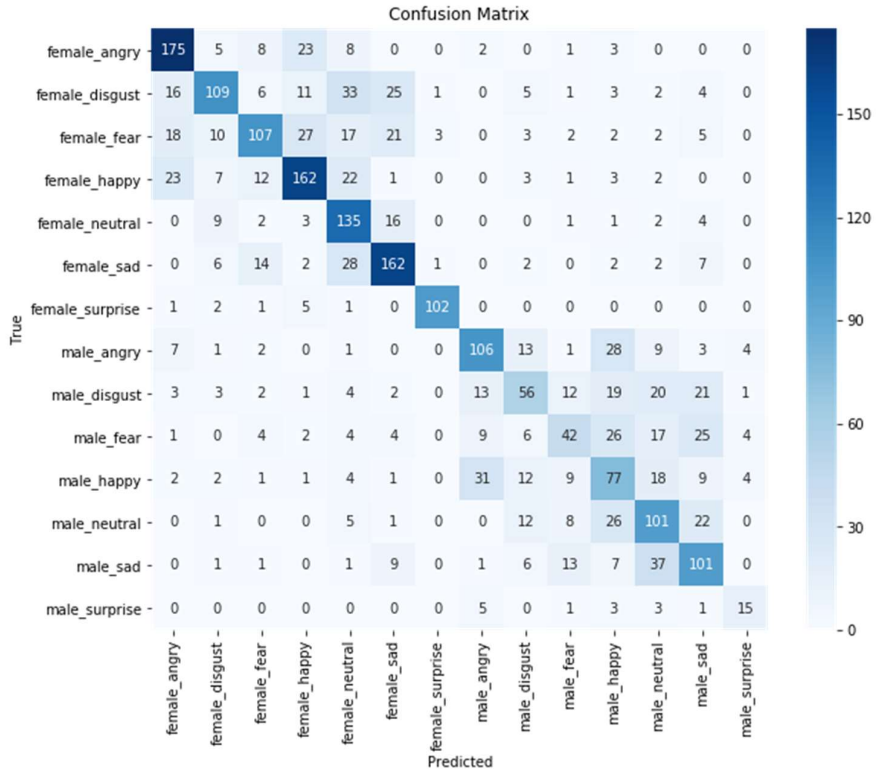
Now let us use the confusion matrix of all the three models and check which model was better in predicting the correct emotion class for the audio files.



Let us take an arbitrary cell of the confusion matrix. Let that cell be (6,6). The above confusion matrix is for the SVM Model. It predicts 72 correct predictions, if we take cell (6,6) into consideration. Now let us see the same cell (6,6) in the confusion matrix of KNN Model:



So here we see that the emotions labels are present. So cell (6,6) can be termed as the (female_sad, female_sad) cell. So, this model does a bit better job as compared to the 72 correct predictions of the SVM model. This model predicts 78 correct predictions. Now let take a look at the confusion matrix of the LSTM Model in the next page:



So here we see that the emotions labels are again present. So cell (6,6) can be termed as the (female_sad, female_sad) cell. So, this LSTM model does a good job as compared to the 72 correct predictions of the SVM model and 78 correct predictions of the KNN Model. This model predicts 162 correct predictions.

Now we understand that the diagonal elements of the matrix mean the correct predictions. So, we add all the 14 diagonal cells and take out how many total correct predictions (out of 2433 total predictions) were made by the respective models.

Lets see for the SVM Model first:

144+82+94+98+96+132+72+92+42+34+45+86+71+11
=1099 total correct predictions

Now if we see the KNN Model:

165+123+103+123+108+131+78+82+43+43+47+77+66+13
=1202 total correct predictions

Lastly we see the LSTM Model:

175+109+107+162+135+162+102+56+42+77+101+101+15
=**1344** total correct predictions

So if we base our understanding on these figures we can see a steady improvement in the no. of total correct predictions from the SVM Model to the KNN Model and then to the LSTM Model. For the figures, the SVM Model recorded an Accuracy of 45.12%, the KNN Model recorded an Accuracy of 49.40% and the LSTM Model recorded an Accuracy of 59.60%.

6.3 Classification Report of the LSTM Model

A classification report is a summary of the performance of a classification model, providing key metrics such as precision, recall, F1-score, and support for each class in the dataset. It is typically used to evaluate the effectiveness of a classification model across multiple classes.

Additionally, the classification report may include macro-average and weighted-average metrics, which aggregate the performance metrics across all classes. Macro-average calculates the unweighted mean of precision, recall, and F1-score across classes, treating each class equally. Weighted-average calculates the weighted mean of precision, recall, and F1-score, where each class's contribution is weighted by its support.

The classification report provides valuable insights into the model's performance for each class, helping to identify strengths and weaknesses and informing decision-making in classification tasks.

Classification Report:

	precision	recall	f1-score	support
0	0.71	0.78	0.74	225
1	0.70	0.50	0.59	216
2	0.67	0.49	0.57	217
3	0.68	0.69	0.68	236
4	0.51	0.78	0.62	173
5	0.67	0.72	0.69	226
6	0.95	0.91	0.93	112
7	0.63	0.61	0.62	175
8	0.47	0.36	0.41	157
9	0.46	0.29	0.36	144
10	0.39	0.45	0.42	171
11	0.47	0.57	0.52	176
12	0.50	0.57	0.53	177
13	0.54	0.54	0.54	28
accuracy			0.60	2433
macro avg	0.60	0.59	0.59	2433
weighted avg	0.60	0.60	0.59	2433

Precision: Precision measures the proportion of correctly predicted instances among all instances predicted as belonging to a particular class. It indicates the accuracy of positive predictions. For instance:

- For class 0: The precision is 0.71, which means that out of all instances predicted as class 0, 71% were correctly classified.
- For class 1: The precision is 0.70, indicating that 70% of instances predicted as class 1 were correct, and so on for other classes.

Recall: Recall (also known as sensitivity) measures the proportion of correctly predicted instances of a particular class among all actual instances of that class. It indicates the ability of the classifier to capture all positive instances. For instance:

- For class 0: The recall is 0.78, indicating that 78% of all actual instances of class 0 were correctly classified.
- For class 1: The recall is 0.50, meaning that only 50% of actual instances of class 1 were correctly classified.

F1-score: F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall, especially when dealing with imbalanced datasets. For instance:

- For class 0: The F1-score is 0.74, which balances precision and recall for class 0.
- For class 1: The F1-score is 0.59, indicating the balance between precision and recall for class 1.

Support: Support represents the number of actual occurrences of each class in the dataset. For instance:

- For class 0: There are 225 instances of class 0 in the dataset.
- For class 1: There are 216 instances of class 1 in the dataset.

Accuracy: Accuracy measures the overall correctness of the classifier across all classes. It is the ratio of correctly classified instances to the total number of instances. In this case, the overall accuracy of the classifier is 0.60, meaning it correctly predicts 60% of the instances.

Macro Avg: Macro average calculates metrics independently for each class and then takes the unweighted mean of the measures. It gives equal weight to each class, regardless of class imbalance.

Weighted Avg: Weighted average calculates metrics for each class independently and then takes the weighted average based on the number of true instances for each class. It gives more weight to classes with more instances.

Now we go classwise to understand these parameters:

Class 0 (female_happy**):**

Precision: 0.71
Recall: 0.78
F1-score: 0.74
Support: 225

For class 0, the model achieves a precision of 0.71, indicating that out of all instances predicted as class 0, 71% were actually class 0. The recall of 0.78 suggests that the model correctly identifies 78% of all actual class 0 instances. The F1-score of 0.74 represents the harmonic mean of precision and recall, providing a balanced measure of the model's performance. The support value of 225 indicates the number of instances in the dataset belonging to class 0.

Class 1 (female_fear**):**

Precision: 0.70
Recall: 0.50
F1-score: 0.59
Support: 216

For class 1, the precision of 0.70 indicates that 70% of the instances predicted as class 1 were correct. However, the recall of 0.50 suggests that only 50% of the actual class 1 instances were correctly identified by the model. The F1-score of 0.59 reflects the balance between precision and recall, while the support value of 216 indicates the number of instances in class 1.

Class 2 (female_sad**):**

Precision: 0.67
Recall: 0.49
F1-score: 0.57
Support: 217

The precision of 0.67 indicates that among all instances predicted as class 2, 67% are actually of class 2. The recall of 0.49 suggests that the model correctly identifies only 49% of all actual instances of class 2. The F1-score of 0.57, which is the harmonic mean of precision and recall, provides a balanced measure of the model's performance for class 2. The support value of 217 indicates the number of actual instances of class 2 in the dataset.

Class 3 (female_disgust**):**

Precision: 0.68
Recall: 0.69
F1-score: 0.68
Support: 236

The precision of 0.68 indicates that among all instances predicted as class 3, 68% are actually of class 3. The recall of 0.69 suggests that the model correctly identifies 69% of all actual instances of class 3. The F1-score of 0.68 provides a balanced measure of the model's performance for class 3. The support value of 236 indicates the number of actual instances of class 3 in the dataset.

Class 4 (female_angry**):**

Precision: 0.51
Recall: 0.78
F1-score: 0.62
Support: 173

The precision of 0.51 indicates that among all instances predicted as class 4, 51% are actually of class 4. The recall of 0.78 suggests that the model correctly identifies 78% of all actual instances of class 4. The F1-score of 0.62 provides a balanced measure of the model's performance for class 4. The support value of 173 indicates the number of actual instances of class 4 in the dataset.

Class 5 (female_neutral**):**

Precision: 0.67
Recall: 0.72
F1-score: 0.69
Support: 226

In class 5, the precision of 0.67 indicates that out of all instances predicted as class 5, 67% were correctly classified, while the recall of 0.72 indicates that 72% of the actual class 5 instances were correctly classified. The F1-score, which is the harmonic mean of precision and recall, is 0.69, suggesting a balanced performance between precision and recall for class 5. The support of 226 indicates the number of instances belonging to class 5.

Class 6 (male_neutral**):**

Precision: 0.95
Recall: 0.91
F1-score: 0.93
Support: 112

For class 6, the precision of 0.95 indicates a high level of precision in classifying instances as class 6, with 95% of the predictions being correct. The recall of 0.91 indicates that 91% of the actual class 6 instances were correctly classified. The high F1-score of 0.93 suggests excellent overall performance for class 6. The support of 112 indicates the number of instances belonging to class 6.

Class 7 (male_sad**):**

Precision: 0.63
Recall: 0.61
F1-score: 0.62
Support: 175

In class 7, the precision of 0.63 indicates that 63% of the instances predicted as class 7 were correct. The recall of 0.61 indicates that 61% of the actual class 7 instances were correctly classified. The F1-score of 0.62 reflects a balanced performance between precision and recall for class 7. The support of 175 indicates the number of instances belonging to class 7 in the dataset.

Class 8 (male_disgust**):**

Precision: 0.47
Recall: 0.36
F1-score: 0.41
Support: 157

For class 8, the precision is 0.47, indicating that 47% of the instances predicted as class 8 were actually members of class 8. The recall is 0.36, meaning that only 36% of the actual instances of class 8 were correctly predicted as such. The F1-score, which combines precision and recall into a single metric, is 0.41. This suggests a moderate balance between precision and recall for class 8.

Class 9 (male_fear**):**

Precision: 0.46
Recall: 0.29
F1-score: 0.36
Support: 144

For class 9, the precision is 0.46, indicating that 46% of the instances predicted as class 9 were actually members of class 9. The recall is 0.29, meaning that only 29% of the actual instances of class 9 were correctly predicted as such. The F1-score for class 9 is 0.36, reflecting a relatively low balance between precision and recall.

Class 10 (male_happy**):**

Precision: 0.39
Recall: 0.45
F1-score: 0.42
Support: 171

For class 10, the precision is 0.39, indicating that 39% of the instances predicted as class 10 were actually members of class 10. The recall is 0.45, meaning that 45% of the actual instances of class 10 were correctly predicted as such. The F1-score for class 10 is 0.42, suggesting a moderate balance between precision and recall, but with room for improvement.

Class 11 (male_angry**):**

Precision: 0.47
Recall: 0.57
F1-score: 0.52
Support: 176

For Class 11, the precision is 0.47, indicating that among all instances predicted as class 11, 47% were actually class 11. The recall for class 11 is 0.57, suggesting that 57% of the actual class 11 instances were correctly identified by the model. The F1-score for class 11 is 0.52, which is the harmonic mean of precision and recall. The support for class 11 is 176, indicating the actual number of instances belonging to class 11 in the dataset.

Class 12 (female_surprise**):**

Precision: 0.50
Recall: 0.57
F1-score: 0.53
Support: 177

The precision for class 12 is 0.50, meaning that 50% of the instances predicted as class 12 were actually class 12. The recall for class 12 is 0.57, indicating that 57% of the actual class 12 instances were correctly identified by the model. The F1-score for class 12 is 0.53, providing a balanced measure of precision and recall for class 12. The support for class 12 is 177, representing the actual number of instances belonging to class 12 in the dataset.

Class 13 (male_surprise)**:**

Precision: 0.54
Recall: 0.54
F1-score: 0.54
Support: 28

The precision for class 13 is 0.54, implying that 54% of the instances predicted as class 13 were actually class 13. The recall for class 13 is 0.54, indicating that 54% of the actual class 13 instances were correctly identified by the model. The F1-score for class 13 is 0.54, representing a balanced measure of precision and recall for class 13. The support for class 13 is 28, indicating the actual number of instances belonging to class 13 in the dataset.

Therefore, the overall model accuracy is 0.60.

6.4 Final Results

1. This project suggests a way to understand the emotion embedded in an audio file.
2. This project talks about various machine learning algorithms and also contains a comparative study among them for accomplishing the above said task.
3. This project successfully completes feature extraction from the audio file using MFCC.
4. This project successfully demonstrates training and testing of models.
5. In the way to understand the emotion embedded in an audio file, this project achieves approximately 60% accuracy.

CHAPTER 7: CONCLUSION

7.1 Limitation of the Project

The project faces various limitations. Firstly, emotions are inherently subjective and can vary widely between individuals. Labelling emotional states based on speech signals alone may be challenging due to the ambiguity and complexity of human emotions. Different annotators may interpret emotions differently, leading to inconsistencies in labelling.

Secondly, speech signals may lack contextual information, such as body language, facial expressions, or situational cues, which can provide valuable context for understanding emotions. Ignoring contextual information may limit the model's ability to accurately recognize emotions in real-world scenarios.

Complex models with a large number of parameters may be prone to overfitting or underfitting, especially when trained on small datasets. These can result in poor generalisation performance and limited robustness to unseen data.

Lastly and most importantly, different languages and cultures express emotions in diverse ways, influencing speech patterns, intonations, and vocabulary choices. Emotion recognition models trained on one language may struggle to generalise to other languages due to these variations. For example, a model trained on English speech data may not perform well when applied to speech samples in languages like Mandarin or Arabic, where emotional expression differs significantly. An inherent limitation in speech emotion recognition is the scarcity of multilingual and culturally diverse datasets. Annotating emotional labels in audio samples from different languages can be challenging due to language-specific nuances and cultural contexts.

7.2 Future Scope

Achieving 60% accuracy indicates room for improvement in the domain, highlighting significant potential for future advancements. Various factors may have contributed to this level of accuracy, including challenges in feature extraction, model complexity, dataset quality, and the inherent complexity of interpreting emotional nuances in audio data. Addressing these issues and exploring innovative techniques could lead to substantial improvements in accuracy, paving the way for more reliable and precise emotion recognition systems.

Using more features could potentially improve accuracy in LSTM-based models for emotion recognition, but it's not guaranteed. Adding more relevant features may help the model capture additional nuances and patterns in the audio data, potentially leading to better performance. However, it's essential to ensure that the additional features are meaningful and contribute positively to the model's predictive power. Additionally, increasing the number of

features can also introduce complexity and may require more data to effectively train the model, as well as careful tuning of hyperparameters to avoid overfitting. Therefore, while adding more features could theoretically enhance accuracy, it's important to balance complexity with performance and consider the overall quality and relevance of the features being used.

The lack of sufficient data or audio files for certain specified classes within the datasets can significantly impact the model's ability to accurately predict those classes. With fewer examples available for training, the model may struggle to learn the distinguishing characteristics and patterns associated with these underrepresented classes. As a result, the model's performance may be compromised, leading to lower accuracy and reliability in predicting instances belonging to these classes. Moreover, the imbalance in class distribution can bias the model towards the majority classes, further exacerbating the issue.

While KNN, SVM, and LSTM are effective, multidimensional models could offer greater insight in speech emotion recognition due to their ability to capture complex relationships and patterns across multiple dimensions, potentially improving accuracy and performance in the project. Using such models to improve upon the accuracy levels could be a promising future aspect in this domain.

In speech emotion recognition, the linguistic diversity of human languages presents a significant challenge to its success. The present project focuses on analysing emotions in English speech, specifically targeting a few accents. However, the potential for incorporating additional languages offers a vast scope for future advancements in the field.

Expanding the project to include a broader range of languages can significantly enhance its applicability and effectiveness. Different languages exhibit unique phonetic characteristics, prosodic features, and cultural nuances that influence the expression and perception of emotions. By incorporating multiple languages, the model can learn to recognize and interpret emotions across diverse linguistic contexts, improving its robustness and generalisation capabilities.

Furthermore, incorporating various accents and dialects within each language adds another layer of complexity and richness to the dataset. Emotions may be expressed differently based on regional variations in speech patterns, intonation, and vocabulary. By encompassing a wide spectrum of linguistic diversity, the model can learn to accommodate these variations and accurately capture the subtle nuances of emotion expression.

Moreover, multilingual speech emotion recognition has practical implications in diverse real-world scenarios. For example, in global communication platforms, customer service applications, or educational settings, understanding and responding to emotions expressed in different languages are crucial for effective interaction and engagement. By developing multilingual models, researchers can address the challenges posed by linguistic diversity and create more inclusive and adaptable systems that cater to a global audience.

While the present project focuses on English speech with specific accents, the incorporation of multiple languages offers immense potential for advancing speech emotion recognition technology. By embracing linguistic diversity and exploring the nuances of emotions across languages and cultures, researchers can pave the way for more comprehensive and effective emotion recognition systems with broader applicability and impact.

Additionally, integrating a frontend module using the Flask framework of python could serve as a valuable future enhancement. This frontend module would provide a user-friendly interface for interacting with the emotion recognition system, allowing users to easily upload audio files, submit them for analysis, and view the predicted emotions. With Flask, developers can quickly build and deploy web applications, enabling seamless integration of the emotion recognition functionality into various platforms and environments. Moreover, a well-designed frontend can enhance user experience, making the system more accessible and intuitive for users of all levels of technical proficiency. By incorporating a frontend module, the emotion recognition system can extend its usability and appeal to a broader audience, fostering adoption and utilisation in diverse contexts such as education, healthcare, and entertainment.

REFERENCES AND CITATIONS

CITATIONS

[1]
Giannakopoulos T (2015) pyAudioAnalysis: An Open-Source Python Library for Audio Signal Analysis. PLoS ONE 10(12): e0144610. doi:10.1371/journal.pone.0144610

[2]
Z. Lu, L. Cao, Y. Zhang, C. -C. Chiu and J. Fan, "Speech Sentiment Analysis via Pre-Trained Features from End-to-End ASR Models," *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Barcelona, Spain, 2020, pp. 7149-7153, doi: 10.1109/ICASSP40776.2020.9052937.

[3]
Kumar T., Mahrishi M., Nawaz S. (2022). A Review of Speech Sentiment Analysis Using Machine Learning. In: Kaiser, M.S., Bandyopadhyay, A., Ray, K., Singh, R., Nagar, V. (eds) Proceedings of Trends in Electronics and Health Informatics. Lecture Notes in Networks and Systems, vol 376. Springer, Singapore. doi: 10.1007/978-981-16-8826-3_3

[4]
Livingstone SR, Russo FA (2018) The Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS): A dynamic, multimodal set of facial and vocal expressions in North American English. PLoS ONE 13(5): e0196391. doi:10.1371/journal.pone.0196391.

[5]
Pichora-Fuller, M. Kathleen; Dupuis, Kate, 2020, "Toronto emotional speech set (TESS)", doi: 10.5683/SP2/E8H2MF, Borealis, V1

[6]
Cao H, Cooper DG, Keutmann MK, Gur RC, Nenkova A, Verma R. CREMA-D: Crowd-sourced Emotional Multimodal Actors Dataset. IEEE Trans Affect Comput. 2014 Oct-Dec;5(4):377-390. doi: 10.1109/TAFFC.2014.2336244. PMID: 25653738; PMCID: PMC4313618.

REFERRENCES

[7]
[i]en.wikipedia.org/wiki/Support_vector_machine
[ii] en.wikipedia.org/wiki/Neural_network

Last Accessed on: 4th April, 2024

[8]
medium.com/@sachinsoni600517/k-nearest-neighbours-introduction-to-machine-learning-algorithms-9dbc9d9fb3b2

Last Accessed on: 4th April, 2024

[9]
https://www.geeksforgeeks.org/adam-optimizer/

Last Accessed on: 4th April, 2024