# Speech Emotion Recognition

*Seasons of Code, 2024*

**Navya Garg**

Mentor: Aakriti Chandra

May 2024 - July 2024

# Chapter 1

# Introduction

This project aims to learn various tools used in machine learning and analysis of large data. From basic implementation of various models to building the final model which predicts the emotion of a person's speech. The project can be found at `https://github.com/GargNavya/Speech-Emotion-Recognition-SoC.git`

We hereby enlist and explain all the tools which were implemented.

# Chapter 2

# Week 1: Basic Python

- There is an assignment for hands-on experience on `numpy`, `pandas` and `matplotlib`.
- There was an introduction to Google Colab too and we saw how to import, edit and push changes to a `.ipynb` file from GitHub.

# Chapter 3

# Week 2: Regression and Classification

## 3.1 Regression

Used to predict a continuous quantity.

## 3.2 Classification

Used for classification in discrete class labels.

## 3.3 Logistic Regression

Logistic regression is a process of modeling the probability of a discrete outcome given an input variable. The most common logistic regression models a binary outcome, which can take two values such as true/false or yes/no.

In this section, we will perform logistic regression on the breast cancer dataset using the `sklearn` library.

### 3.3.1 Importing Libraries

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import datasets
import numpy as np
```

### 3.3.2 Preparing Data

We load the breast cancer dataset, split the data into training and testing sets, and standardize the features.

```
breast_cancer = datasets.load_breast_cancer()
X, y = breast_cancer.data, breast_cancer.target

# Splitting data for training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.2, random_state=1234)

# Standardizing the data
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

### 3.3.3 Implementing Logistic Regression

We define the sigmoid function and the logistic regression class with methods for fitting the model and predicting outcomes.

```
# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Logistic Regression class
class LogisticRegression:
    def __init__(self, lr=0.01, iters=1000):
        self.lr = lr
        self.iters = iters

    def fit(self, X, y):
        m, n = X.shape
        w = np.zeros(n)
        b = 0
        for i in range(self.iters):
            y_pred = sigmoid(np.dot(X, w) + b)
            dw = np.mean((np.dot((y_pred - y), X)), axis=0)
            db = np.mean(y_pred - y)
            w = w - dw * self.lr
            b = b - db * self.lr
        self.w = w
        self.b = b

    def predict(self, X):
        self.probability = sigmoid(np.dot(X, self.w) + self.b)
        y_pred = np.where(self.probability > 0.5, 1, 0)
        return y_pred
```

### 3.3.4 Training the Model and Making Predictions

We create an instance of the `Logistic Regression` class, fit the model to the training data, and make predictions on the test data.

```
model = LogisticRegression(lr=0.01, iters=50000)
model.fit(X_train, y_train)
print("Predicted␣values:\n", model.predict(X_test))
print("Actual␣values:\n", y_test)
```

### 3.3.5 Calculating Accuracy

We calculate the accuracy of the model by comparing the predicted values with the actual values.

```
matches = y_test == model.predict(X_test)
print("Accuracy␣of␣the␣model:␣", (np.sum(matches) / len(y_test)) * 100,
    "%", sep="")
```

### 3.3.6 Binary Cross Entropy Loss

We define a function to calculate the binary cross entropy loss for the test data.

```
def BCELoss(y, y_pred):
    return -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))

print("Value␣of␣cost␣function:", BCELoss(y_test, model.probability))
```

## 3.4 Support Vector Machine (SVM) from Scratch

### 3.4.1 SVM Basic Concepts and Equations

**Hyperplane**: A decision boundary separating different classes. In an $n$-dimensional space, the hyperplane can be represented as:

$$w \cdot x + b = 0$$

where $w$ is the weight vector, $x$ is the input vector, and $b$ is the bias term.

**Margin**: The distance between the hyperplane and the nearest data point of each class. SVM aims to maximize this margin.

**Optimization Problem**: SVM solves the following optimization problem to find the optimal hyperplane:

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

subject to:

$$y_i(w \cdot x_i + b) \geq 1 \quad \forall i$$

where $y_i$ is the class label of the $i$-th data point, and $x_i$ is the $i$-th data point.

### 3.4.2 Objective Function (Optimized Version)

The objective function of a linear SVM with L2 regularization is given by:

$$\min_{w,b} \left[ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(w \cdot x_i - b)) \right]$$

Where: - $w$ is the weight vector. - $b$ is the bias term. - $C$ is the regularization parameter. - $y_i$ are the labels {-1, 1}. - $x_i$ are the feature vectors. - $n$ is the number of samples.

### 3.4.3 Hinge Loss

The hinge loss for a single sample is:

$$L_i(w, b) = \max(0, 1 - y_i(w \cdot x_i - b))$$

### 3.4.4 Gradient of the Objective Function

To minimize the objective function, we use gradient descent. We need the gradients of the objective function with respect to $w$ and $b$.

#### Gradient with Respect to $w$

The gradient of the regularization term is:

$$\nabla_w \left( \frac{1}{2}\|w\|^2 \right) = w$$

The gradient of the hinge loss term is:

$$\nabla_w L_i(w, b) = \begin{cases} 0 & \text{if } y_i(w \cdot x_i - b) \geq 1 \\ -y_i x_i & \text{if } y_i(w \cdot x_i - b) < 1 \end{cases}$$

Combining these, the total gradient with respect to $w$ is:

$$\nabla_w J(w, b) = w + C\sum_{i=1}^{n} \nabla_w L_i(w, b)$$

For a single sample update (stochastic gradient descent), this becomes:

$$\nabla_w J(w, b) = w$$

if $y_i(w \cdot x_i - b) \geq 1$ and

$$\nabla_w J(w, b) = w - C y_i x_i$$

if $y_i(w \cdot x_i - b) < 1$

**Gradient with Respect to $b$**

The gradient of the hinge loss term with respect to $b$ is:

$$\nabla_b L_i(w,b) = \begin{cases} 0 & \text{if } y_i(w \cdot x_i - b) \geq 1 \\ y_i & \text{if } y_i(w \cdot x_i - b) < 1 \end{cases}$$

For a single sample update, this becomes:

$$\nabla_b J(w,b) = 0$$

if $y_i(w \cdot x_i - b) \geq 1$ and

$$\nabla_b J(w,b) = -Cy_i$$

if $y_i(w \cdot x_i - b) < 1$

### 3.4.5 Weight and Bias Update Equations

Using gradient descent, we update the weights and bias by moving in the direction of the negative gradient:

1. **Condition is met (correct classification with a margin):**

$$y_i(w \cdot x_i - b) \geq 1$$

$$w \leftarrow w - \eta \nabla_w J(w,b) = w - \eta(2\lambda w)$$

2. **Condition is not met (misclassification or within the margin):**

$$y_i(w \cdot x_i - b) < 1$$

$$w \leftarrow w - \eta \nabla_w J(w,b) = w - \eta(2\lambda w - y_i x_i)$$
$$b \leftarrow b - \eta \nabla_b J(w,b) = b - \eta(-y_i) = b + \eta y_i$$

### 3.4.6 SVM Implementation in Python

```
# Importing the libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the dataset
data = pd.read_csv('data.csv') #data.csv given in the week2 folder

# Dataset has features in columns 'feature1', 'feature2' ... and the
    target in 'target'
X = data[['feature1', 'feature2']].values
y = data['target'].values
```

```
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)

class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters
        =1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        # Initialize weights and bias
        w = np.array([1,1])
        b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                # Check for the condition and do weight and bias
                    updation specified above
                check_value_of = y[idx]*(np.dot(x_i, w) + b)
                if check_value_of >= 1:
                    w = w - self.lr * (2 * self.lambda_param * w)
                else:
                    w = w - self.lr * (2*self.lambda_param*w - y[idx]*
                        x_i)
                    b = b + self.lr * y[idx]

        self.w = w
        self.b = b

    def predict(self, X):
        # Predict to test SVM on X_test
        classifying_value = np.dot(X, self.w) - self.b
        y_pred = np.where(classifying_value >= 1, 1, -1)
        return y_pred

# Initialize and train the SVM
myfirstSVM = SVM(learning_rate=0.001, lambda_param=0.01, n_iters=1000)
myfirstSVM.fit(X_train, y_train)

# Predictions
y_pred = myfirstSVM.predict(X_test)

# Evaluate the performance using accuracy_score function imported from
    sklearn.metrics
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

## 3.5 Summary

The topics covered in this week were:

- Linear Regression
- Logistic Regression
- Ridge Regression
- Naive Bayes Classifier
- Support Vector Machines (SVM)

# Chapter 4

# Week 3: Neural Networks

This week we learned about Neural Networks in detail, working out all the maths behind backpropagation and parameter-updation.

## 4.1 Neural Network

### 4.1.1 Architecture

The structure of the neural network is outlined as follows, where $X$ represents the input, $A^{[0]}$ denotes the first layer, $Z^{[1]}$ signifies the unactivated layer 1, $A^{[1]}$ stands for the activated layer 1, and so forth. The weights and biases are represented by $W$ and $b$ respectively:

$$\begin{aligned} A^{[0]} &= X \\ Z^{[1]} &= W^{[1]}A^{[0]} + b^{[1]} \\ A^{[1]} &= \text{ReLU}(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= \text{softmax}(Z^{[2]}) \end{aligned}$$

The loss is computed using the cross-entropy loss function:

$$\text{Loss} = \text{cross-entropy-loss}(A^{[2]})$$

### 4.1.2 Activation Functions

#### ReLU

The ReLU activation function is used for the hidden layer:

$$\text{ReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

```
def ReLU(Z):
    return np.where(Z > 0, Z, 0)
```

**Softmax**

The softmax function is used for the output layer:

$$\text{softmax}(z) = \frac{e^z}{\sum_{i=1}^{10} e^z}$$

```
def softmax(Z):
    exp_Z = np.exp(Z)
    return exp_Z / np.sum(exp_Z, axis=1, keepdims=True)
```

### 4.1.3 Class Implementation

We define a neural network class to encapsulate all functionalities.

```
class NN:
    def __init__(self, input_size, hidden_size, output_size,
        learning_rate=0.01):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        self.W1 = np.random.randn(hidden_size, input_size) * 0.01
        self.B1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(output_size, hidden_size) * 0.01
        self.B2 = np.zeros((1, output_size))

    def forward_propagation(self, X):
        self.A0 = X
        self.Z1 = np.matmul(self.A0, self.W1.T) + self.B1
        self.A1 = ReLU(self.Z1)
        self.Z2 = np.matmul(self.A1, self.W2.T) + self.B2
        self.A2 = softmax(self.Z2)

    def one_hot(self, y):
        temp = np.zeros((y.size, 10))
        for i in range(y.size):
            temp[i][y[i]] = 1
        return temp

    def backward_propagation(self, X, y):
        m = X.shape[0]
        self.dZ2 = self.A2 - y
        self.dW2 = (1/m) * np.matmul(self.dZ2.T, self.A1)
        self.dB2 = (1/m) * np.sum(self.dZ2, axis=0, keepdims=True)
        self.dZ1 = (np.matmul(self.dZ2, self.W2)) * np.where(self.Z1 >=
            0, 1, 0)
        self.dW1 = (1/m) * np.matmul(self.dZ1.T, X)
        self.dB1 = (1/m) * np.sum(self.dZ1, axis=0, keepdims=True)

    def update_params(self):
        self.W2 -= self.learning_rate * self.dW2
        self.B2 -= self.learning_rate * self.dB2
        self.W1 -= self.learning_rate * self.dW1
        self.B1 -= self.learning_rate * self.dB1

    def get_predictions(self, X):
        self.forward_propagation(X)
        return np.argmax(self.A2, axis=1)

    def get_accuracy(self, X, y):
        predictions = self.get_predictions(X)
        return np.mean(predictions == y) * 100

    def gradient_descent(self, X, y, iters=1000):
        Y = self.one_hot(y)
        for i in range(iters):
            self.forward_propagation(X)
            self.backward_propagation(X, Y)
            self.update_params()
            if i % (iters/10) == 0:
                print(f"Iteration␣{i}:␣Cost␣=␣{self.cross_entropy_loss(
                    X,␣y)}")

    def cross_entropy_loss(self, X, y):
        Y = self.one_hot(y)
        self.forward_propagation(X)
        return -np.mean(np.sum(Y * np.log(self.A2), axis=1))

    def show_predictions(self, X, y, num_samples=10):
        random_indices = np.random.randint(0, X.shape[0], size=
           num_samples)
        for index in random_indices:
            sample_image = X[index, :].reshape((28, 28))
            plt.imshow(sample_image, cmap='gray')
            plt.title(f"Actual:␣{y[index]},␣Predicted:␣{self.
               get_predictions(X[index:index+1])[0]}")
            plt.show()
```

### 4.1.4 Training the Model

We split the dataset into training and testing data, normalize the data, and train the model.

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

miu = np.mean(X_train, axis=(0, 1), keepdims=True)
stds = np.std(X_train, axis=(0, 1), keepdims=True)
mius = np.mean(X_test, axis=(0, 1), keepdims=True)
```

```
stdse = np.std(X_test, axis=(0, 1), keepdims=True)

X_normal_train = (X_train - miu) / (stds + 1e-7)
X_normal_test = (X_test - mius) / (stdse + 1e-7)

X_normal_train = X_normal_train.reshape((60000, -1))
X_normal_test = X_normal_test.reshape((10000, -1))

model = NN(input_size=784, hidden_size=128, output_size=10,
   learning_rate=0.5)
model.gradient_descent(X_normal_train, Y_train, iters=100)
print(f"Accuracy:␣{model.get_accuracy(X_normal_test,␣Y_test)}%")
```

### 4.1.5 Results

After training the model, we achieved an accuracy of approximately 94.24% on the test dataset. The cost function was monitored throughout the training process to ensure correct model behaviour.

## 4.2 RNN

Time series forecasting is an important topic in Machine Learning. In this report, we forecast sunspot data from January 1749 to December 1983 using a Recurrent Neural Network (RNN). We split the data into training and testing sets, with 80% of the data for training and 20% for testing. The data is scaled to the range [0,1] using the `MinMaxScaler` from `sklearn.preprocessing`.

### 4.2.1 Data Preparation

The data is accessed from a given URL and processed as follows:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

def get_train_test(url):
    data = pd.read_csv(url)
    data['Month'] = pd.to_datetime(data['Month'])
    data.set_index('Month', inplace=True)
    data.sort_index(inplace=True)

    train_len = int(0.8 * data.shape[0])
    train, test = data.iloc[:train_len], data.iloc[train_len:]

    scaler = MinMaxScaler(feature_range=(0,1))
    train_scaled = scaler.fit_transform(train).reshape(-1)
    test_scaled = scaler.fit_transform(test).reshape(-1)
    return train_scaled, test_scaled
```

```
sunspots_url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/
   master/monthly-sunspots.csv'
train_data, test_data = get_train_test(sunspots_url)

X_train = train_data[:-1]
y_train = train_data[1:]

X_test = test_data[:-1]
y_test = test_data[1:]
```

### 4.2.2 Neural Network

#### Architecture

The RNN model consists of a layer with 64 nodes followed by a Feed Forward Neural Network with one output node. The ReLU function is used for activation in the hidden layer, and the Sigmoid function is used for the output layer.

#### Activation Functions

**ReLU**
The ReLU activation function is defined as:

$$\text{ReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

```
def ReLU(Z):
    return np.where(Z > 0, Z, 0)
```

**Sigmoid**
The Sigmoid activation function is defined as:

$$\text{Sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

```
def Sigmoid(Z):
    return 1 / (1 + np.exp(-Z))
```

#### RNN Implementation

The RNN model is implemented as follows:

```
class RNN:
    def __init__(self, n_nodes=64, learning_rate=0.01):
        self.n_nodes = n_nodes
        self.lr = learning_rate
        self.w = np.random.rand()
        self.u = np.random.rand()
        self.b1 = 0
        self.v = np.random.rand()
        self.b2 = 0

        self.S0 = None
        self.Z1 = None
        self.S1 = None
        self.Z2 = None
        self.Y_pred = None

        self.dZ2 = None
        self.dv = None
        self.db2 = None
        self.dS1 = None
        self.dZ1 = None
        self.dw = None
        self.du = None
        self.db1 = None

    def forward_propagation(self, X):
        m = X.shape[1]
        self.S0 = np.zeros(m)

        for i in range(self.n_nodes):
            self.x = X[i]
            if i != 0:
                self.S0 = self.S1

            self.Z1 = self.w * self.S0 + self.u * self.x + self.b1
            self.S1 = ReLU(self.Z1)

        self.Z2 = self.v * self.S1 + self.b2
        self.Y_pred = Sigmoid(self.Z2)

    def cost(self, y):
        m = len(self.x)
        return (1/m) * np.sum(np.power(self.Y_pred - y, 2))

    def backward_propagation(self, X, y):
        m = X.shape[1]

        self.dZ2 = 2 * (self.Y_pred - y) * (1 - self.Y_pred) * self.
           Y_pred
        self.dv = (1/m) * np.matmul(self.dZ2, self.S1.T)
        self.db2 = (1/m) * np.sum(self.dZ2)
        self.dS1 = self.v * self.dZ2
        self.dZ1 = self.dS1 * np.where(self.Z1 > 0, 1, 0)
        self.dw = (1/m) * np.matmul(self.dZ1, self.S0.T)
        self.du = (1/m) * np.matmul(self.dZ1, self.x)
        self.db1 = (1/m) * np.sum(self.dZ1)
    def update_params(self):
        self.w -= self.lr * self.dw
        self.u -= self.lr * self.du
        self.b1 -= self.lr * self.db1
        self.v -= self.lr * self.dv
        self.b2 -= self.lr * self.db2

    def gradient_descent(self, input_seq, output_seq, iters=1000):
        batch_size = self.n_nodes
        batches = [input_seq[i:i+batch_size] for i in range(len(
           input_seq) - batch_size + 1)]
        X = np.array(batches).T
        y = output_seq[self.n_nodes - 1:]

        for i in range(iters):
            self.forward_propagation(X)
            self.backward_propagation(X, y)
            self.update_params()

            if i % (iters/10) == 0:
                print(f"Iteration␣{i}:␣Cost␣=␣{self.cost(y)}")

    def get_predictions(self, X):
        n = len(X)
        Y_pred = []
        s = 0
        for i in range(n):
            x = X[i]

            z1 = self.w * s + self.u * x + self.b1
            s = ReLU(z1)
            y_pred = Sigmoid(self.v * s + self.b2)
            Y_pred.append(y_pred)

        return Y_pred
```

### Training the Model

The model is trained on the training dataset:

```
model = RNN(n_nodes=64, learning_rate=0.01)
model.gradient_descent(input_seq=X_train, output_seq=y_train, iters
   =100000)
```

### Predictions

The trained model is used to make predictions on the training and testing datasets:

```
train_predict = model.get_predictions(X_train)
test_predict = model.get_predictions(X_test)
```

### 4.2.3 Results

The Root Mean Squared Error (RMSE) is calculated for both training and testing data:

```
import math
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

train_rmse = math.sqrt(mean_squared_error(y_train, train_predict))
test_rmse = math.sqrt(mean_squared_error(y_test, test_predict))

print('Train␣RMSE:␣%.3f␣RMSE' % train_rmse)
print('Test␣RMSE:␣%.3f␣RMSE' % test_rmse)
```

The RMSE values are:

- Train RMSE: 0.072
- Test RMSE: 0.085

## 4.3 Image preprocessing with `librosa`

In this assignment, we will explore the power of the Librosa library and demonstrate how a single line of code can perform wonders in audio analysis. Before running the code, ensure that Librosa is installed using the command `pip install librosa`.

### 4.3.1 Code Implementation

#### Loading and Displaying Audio Information

First, we load the audio file and display its sample rate and duration.

```
import librosa

file_path = 'bird_sound.wav'

# Load the audio file giving the file_path as argument
y, sr = librosa.load(file_path)

# Display the sample rate and duration (use librosa.get_duration())
print("Sample␣Rate:", sr)
print("Duration:", librosa.get_duration(y=y, sr=sr))
```

Output:

```
Sample Rate: 22050
Duration: 145.85024943310657
```

### Playing the Audio File

To play the audio file, uncomment the following code:

```
from IPython.display import Audio
Audio(data=y, rate=sr)
```

### Plotting the Waveform

Next, we plot the waveform of the audio file.

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(14,5))
# Plot the waveform
plt.plot(np.linspace(0, librosa.get_duration(y=y, sr=sr), len(y)), y)

plt.xlabel('Time␣(s)')
plt.ylabel('Amplitude')
plt.show()
```

### Computing and Plotting the Spectrogram

We compute the Short-Time Fourier Transform (STFT) and plot the spectrogram.

```
# Compute the Short-Time Fourier Transform (STFT)
D = librosa.stft(y)

# Convert the amplitude to dB
dB = librosa.amplitude_to_db(np.abs(D), ref=np.max)

plt.figure(figsize=(14, 5))
# Plot the spectrogram
librosa.display.specshow(dB, sr=sr, x_axis='time', y_axis='log', cmap='
    viridis')

plt.colorbar(format='%+2.0f␣dB')
plt.title('Spectrogram')
plt.xlabel('Time␣(s)')
plt.ylabel('Frequency␣(Hz)')
plt.show()
```

### Extracting Audio Features

We extract various audio features such as zero crossing rate, spectral centroids, and Mel-Frequency Cepstral Coefficients (MFCCs).

### Zero Crossing Rate

```
# Zero crossing rate
zcr = librosa.feature.zero_crossing_rate(y)

plt.figure(figsize=(14,5))
# Plot zero crossing rate
frames = range(len(zcr[0]))
plt.plot(frames, zcr[0])

plt.title("Zero␣Crossing␣Rate")
plt.xlabel(’Frames’)
plt.ylabel(’Rate’)
plt.show()
```

### Spectral Centroids

```
# Spectral centroids
spectral_centroids = librosa.feature.spectral_centroid(y=y, sr=sr)[0]

# Plot spectral centroids
frames = range(len(spectral_centroids))
t = librosa.frames_to_time(frames)
plt.figure(figsize=(14, 5))
librosa.display.waveshow(y, sr=sr, alpha=0.4)
plt.plot(t, librosa.util.normalize(spectral_centroids), color=’r’)
plt.title(’Spectral␣Centroid’)
plt.xlabel(’Time␣(s)’)
plt.ylabel(’Normalized␣Centroid’)
plt.show()
```

### Mel-Frequency Cepstral Coefficients (MFCCs)

```
# Mel-Frequency Cepstral Coefficients (MFCCs)
mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)

plt.figure(figsize=(14, 5))
# Plot MFCCs
librosa.display.specshow(mfccs, sr=sr, x_axis=’time’)
plt.colorbar(format=’%+2.0f␣dB’)
plt.title(’MFCCs’)
plt.xlabel(’Time␣(s)’)
plt.ylabel(’MFCC␣Coefficients’)
plt.show()
```

### Time-Stretching the Audio

We apply time-stretching to the audio and compare the original and time-stretched waveforms.

```
# Time-stretching the audio
factor = 0.5
y_stretched = librosa.effects.time_stretch(y=y, rate=factor)

# Plot original and time-stretched waveform
plt.figure(figsize=(14, 5))
plt.subplot(2, 2, 1)
librosa.display.waveshow(y, sr=sr)
plt.title('Original␣Waveform')
plt.subplot(2, 1, 2)
librosa.display.waveshow(y_stretched, sr=sr)
plt.title('Time-Stretched␣Waveform')
plt.xlabel('Time␣(s)')
plt.ylabel('Amplitude')
plt.show()
```

### Pitch Shifting the Audio

Finally, we apply pitch shifting to the audio and compare the original and pitch-shifted waveforms.

```
# Pitch shifting the audio
y_shifted = librosa.effects.pitch_shift(y=y, sr=sr, n_steps=5)

# Plot original and pitch-shifted waveform
plt.figure(figsize=(14, 5))
plt.subplot(2, 1, 1)
librosa.display.waveshow(y, sr=sr)
plt.title('Original␣Waveform')
plt.subplot(2, 1, 2)
librosa.display.waveshow(y_shifted, sr=sr)
plt.title('Pitch-Shifted␣Waveform')
plt.xlabel('Time␣(s)')
plt.ylabel('Amplitude')
plt.show()
```

# Chapter 5

# Audio Classification using TensorFlow and Librosa

This chapter explores the process of audio classification using TensorFlow and Librosa. We will go through the steps of feature extraction, data preparation, model training, and evaluation. The objective is to build a model that can classify audio into different emotion categories based on extracted features.

## 5.1 Feature Extraction

Feature extraction is a fundamental step in audio classification. We use the Librosa library to extract meaningful features from audio files. The features we extract include Mel-frequency cepstral coefficients (MFCCs), chroma features, and mel-spectrograms. Below is the function used to extract these features:

```
import librosa
import numpy as np

def extract_features(file_path):
    # Load the audio file
    audio, sr = librosa.load(file_path)

    # Extract MFCCs
    mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13)
    mfccs = np.mean(mfccs.T, axis=0)

    # Extract chroma features
    stft = np.abs(librosa.stft(audio))
    chroma = librosa.feature.chroma_stft(S=stft, sr=sr)
    chroma = np.mean(chroma.T, axis=0)
    # Extract mel-spectrogram
    melspec = librosa.feature.melspectrogram(y=audio, sr=sr)
    melspec = np.mean(melspec.T, axis=0)

    return mfccs, chroma, melspec
```

## 5.2 Loading and Preparing Data

We load the dataset of audio files, extract features from each file, and prepare the data for training and testing. The dataset is split into training and testing sets. The following function demonstrates this process:

```
import os
from sklearn.model_selection import train_test_split

def one_hot(y):
    temp = np.zeros(8)
    temp[y] = 1
    return temp

def load_data(test_size):
    features = []
    labels = []

    for i in range(1, 25):
        I = "{:02}".format(i)
        audio_dir_path = f'./Audio_Speech/Actor_{I}'

        for file_name in os.listdir(path=f'./Audio_Speech/Actor_{I}'):
            file_path = os.path.join(audio_dir_path, file_name)
            mfccs, chroma, mel = extract_features(file_path)

            feature_row = np.hstack([mfccs, chroma, mel])
            features.append(feature_row)

            label = one_hot(int(file_name.split('-')[3]))
            labels.append(label)

    features = np.array(features)
    labels = np.array(labels)

    # Splitting data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=t
    return X_train, X_test, y_train, y_test
```

## 5.3 Model Definition and Training

We define a simple neural network model using TensorFlow. The model consists of one hidden layer with 300 nodes and an output layer that corresponds to the number of emotion classes. The model is compiled with categorical cross-entropy loss and the Adam optimizer. Here's the code for defining and training the model:

```
import tensorflow as tf

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(300, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(y_train.shape[1], activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy']

# Fit the model
history = model.fit(
    X_train,
    y_train,
    epochs=300,
    batch_size=256,
    validation_split=0.2
)
```

## 5.4 Model Evaluation

After training the model, we evaluate its performance on the test set. We use accuracy as the evaluation metric and also generate a classification report to assess the model's performance across different emotion categories. The code for evaluation is as follows:

```
from sklearn.metrics import classification_report
import numpy as np

# Predict on the test data
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

class_names = ['Neutral', 'Calm', 'Happy', 'Sad', 'Angry', 'Fearful', 'Disgust', 'Sur
report = classification_report(y_test_classes, y_pred_classes, target_names=class_nam
print(report)
```

## 5.5 Conclusion

In this chapter, we demonstrated how to perform audio classification using TensorFlow and Librosa. We covered feature extraction, data preparation, model training, and evaluation. The approach illustrated can be adapted for various audio classification tasks and serves as a foundation for more complex audio analysis applications.