Audio MNIST Digit Recognition

Context

In the past decades, significant advances have been achieved in the area of audio recognition and a lot of research is going on globally to recognize audio data or speech using Deep Learning. The most common use case in this field is converting audio to spectrograms and vice versa.

Audio in its raw form is usually a wave and to capture that using a data structure, we need to have a huge array of amplitudes even for a very short audio clip. Although it depends on the sampling rate of the sound wave, this structured data conversion for any audio wave is very voluminous even for low sampling rates. So it becomes a problem to store and computationally very expensive to do even simple calculations on such data.

One of the best economical alternatives to this is using spectrograms. Spectrograms are created by doing Fourier or Short Time Fourier Transforms on sound waves. There are various kinds of spectrograms but the ones we will be using are called MFCC spectrograms. To put it in simple terms, a spectrogram is a way to visually encapsulate audio data. It is a graph on a 2-D plane where the X-axis represents time and the Y-axis represents Mel Coefficients. But since it is continuous on a 2-D plane, we can treat this as an image.

Objective

The objective here is to build an Artificial Neural Network that can look at Mel or MFCC spectrograms of audio files and classify them into 10 classes. The audio files are recordings of different speakers uttering a particular digit and the corresponding class to be predicted is the digit itself.

Dataset

The dataset we will use is the **Audio MNIST dataset**, which has audio files (having .wav extension) stored in 10 different folders. Each folder consists of these digits spoken by a particular speaker.

Understanding the required packages

- `Librosa` : Librosa is a Python package that helps in dealing with audio data. **librosa.display** visualizes and displays the audio data using Matplotlib. Similarly, there exists a collection of submodules under librosa that provides various other functionalities. **Run the command in the below cell to install the library**.
- `IPython.display` : Display is a public API to display the tools available in Ipython. In this case study, we will create an audio object to display the digits in the MNIST audio data.
- `tqdm` : tqdm is a Python package that allows us to add a progress bar to our application. This package will help us in iterating over the audio data.

Importing the necessary libraries and loading the data

In [1]:
```
# For Audio Preprocessing
import librosa
import librosa.display as dsp
from IPython.display import Audio

# For Data Preprocessing
import pandas as pd
import numpy as np
import os

# For Data Visualization
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm


#The data is provided as a zip file
import zipfile
import os
```

In [2]:
```
sns.set_style("dark")
```

Check some of the audio samples

The below function called "get_audio" takes a digit as an argument and plots the audio wave and returns the audio for a given digit.

Let's understand the functioning of some of the new functions used to create the get_audio() function.

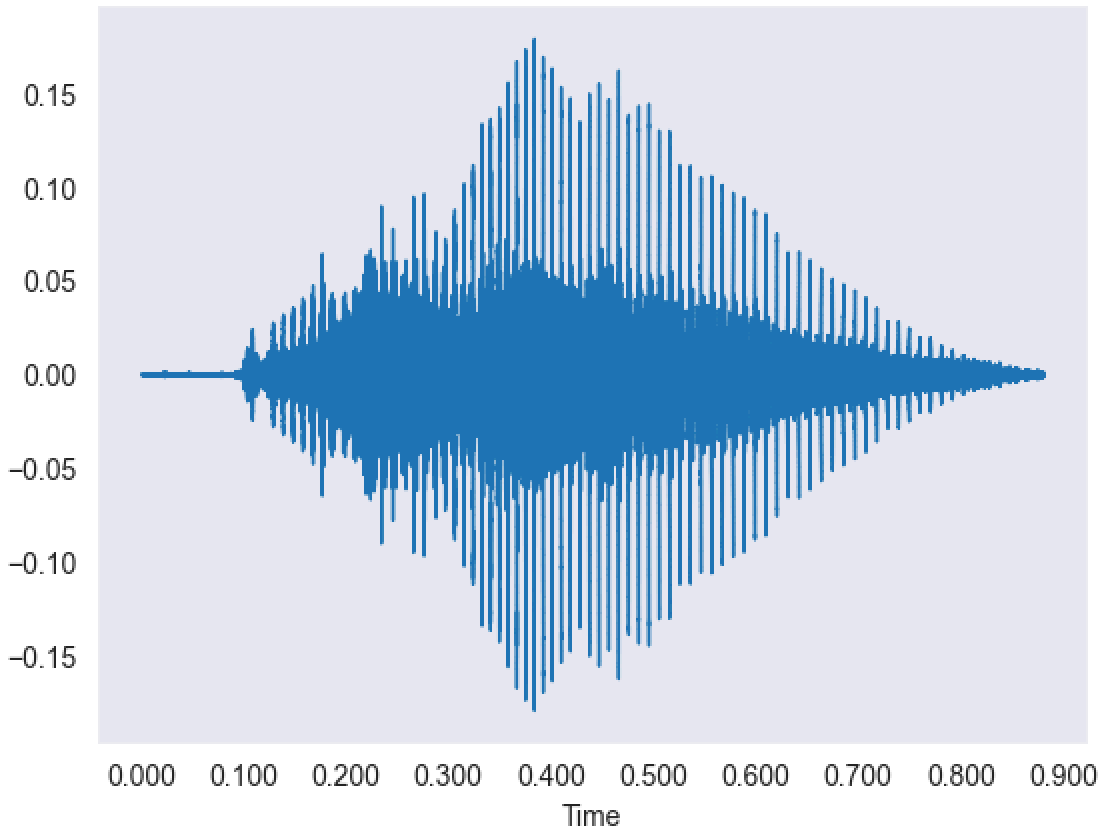
- `.wav` : .wav is a file format like .csv which stores the raw audio format. We will load the .wav file using the librosa package.

- `dsp.waveshow()` : It visualizes the waveform in the time domain. This method creates a plot that alternates between a raw samples-based view of the signal and an amplitude-envelope view of the signal. The "sr" parameter is the sampling rate, i.e., samples per second.
- `Audio()` : From the Ipython package, we can create an audio object.

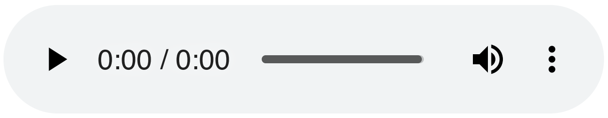
In [33]:
```
def get_audio(digit = 0):

    # Audio Sample Directory
    sample = np.random.randint(1, 10)

    # Index of Audio
    index = np.random.randint(1, 5)

    # Modified file location
    if sample < 10:
        file = f"data/0{sample}/{digit}_0{sample}_{index}.wav"

    else:
        file = f"data/{sample}/{digit}_{sample}_{index}.wav"


    # Get Audio from the location
    # Audio will be automatically resampled to the given rate (default sr =
    data, sample_rate = librosa.load(file)

    # Plot the audio wave
    dsp.waveshow(data, sr = sample_rate)
    plt.show()

    # Show the widget
    return Audio(data = data, rate = sample_rate)
```

```
The history saving thread hit an unexpected error (OperationalError('attempt
to write a readonly database')).History will not be written to the database.
```

In [4]:
```
# Show the audio and plot of digit 0
get_audio(0)
```

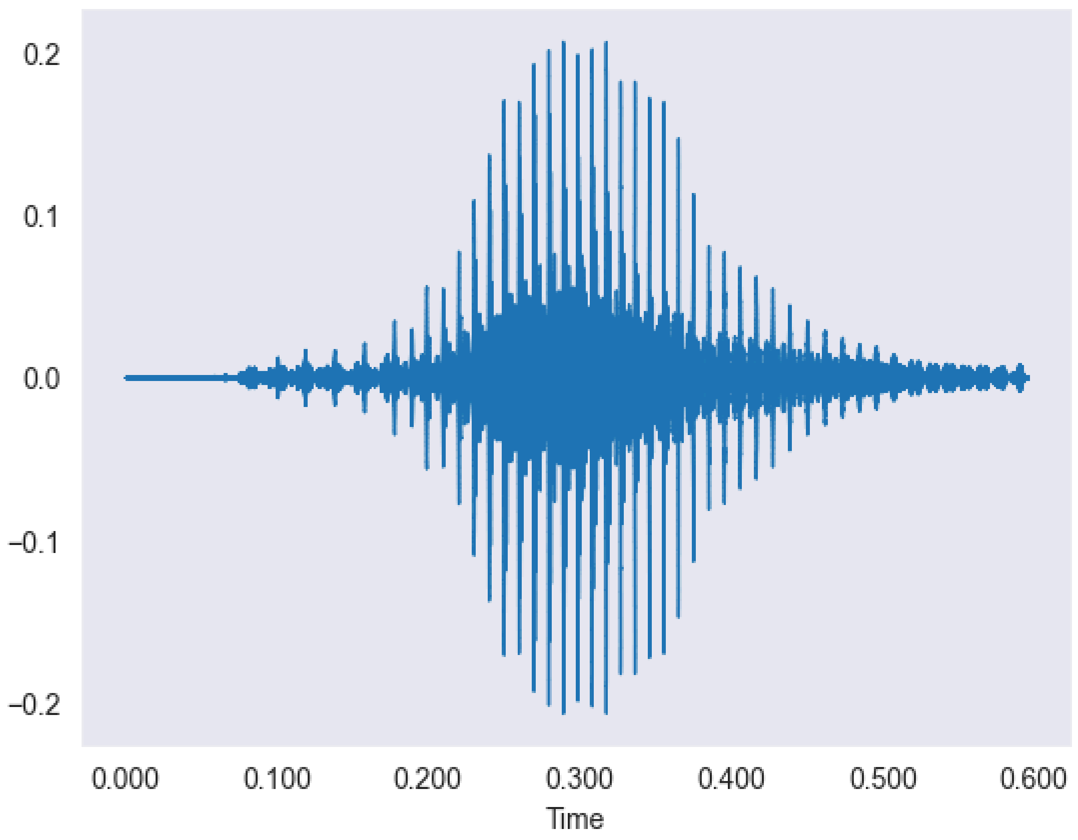


Out[4]:

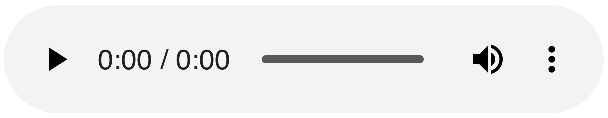


In [5]:

```
# Show the audio and plot of digit 1
get_audio(1)
```

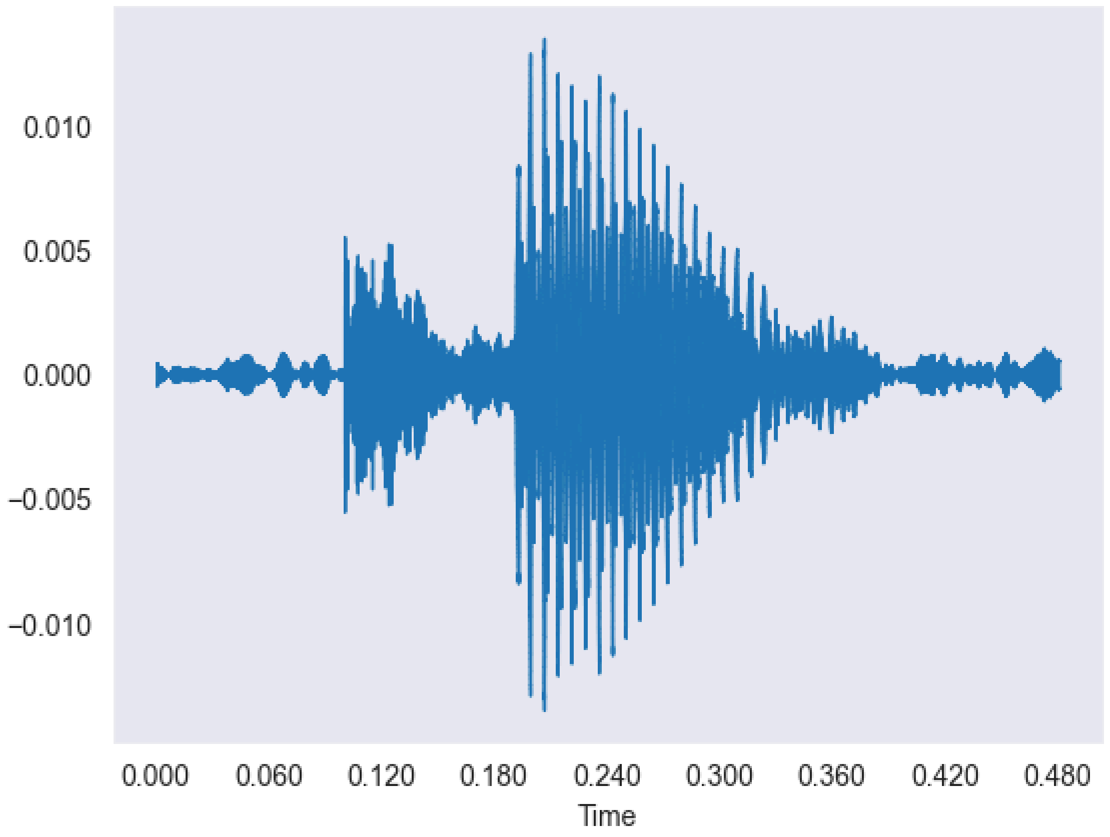


Out[5]:

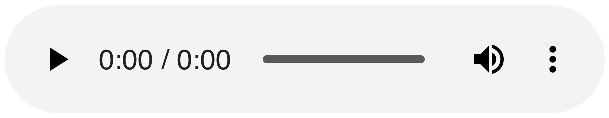


In [6]:

```
# Show the audio and plot of digit 2
get_audio(2)
```

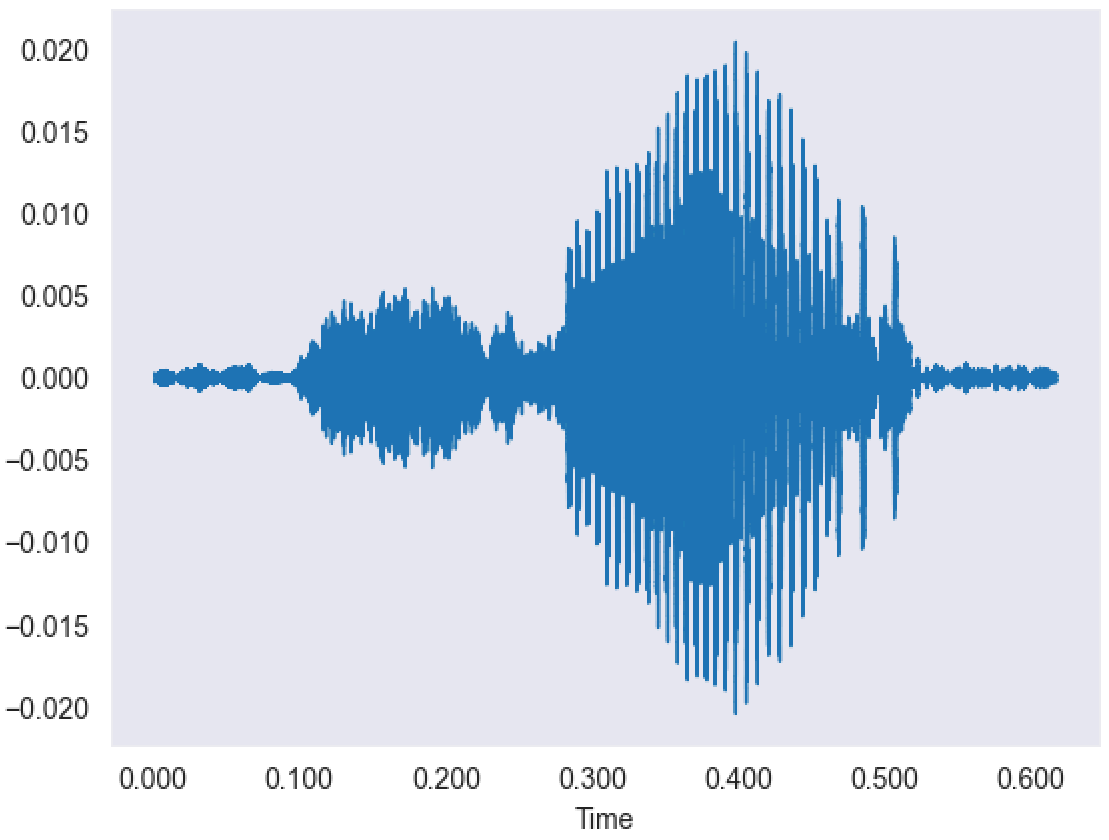


Out[6]:

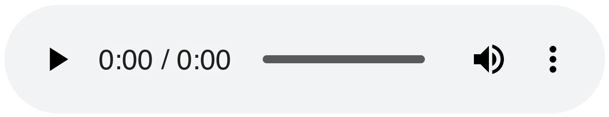


In [7]:

```
# Show the audio and plot of digit 3
get_audio(3)
```

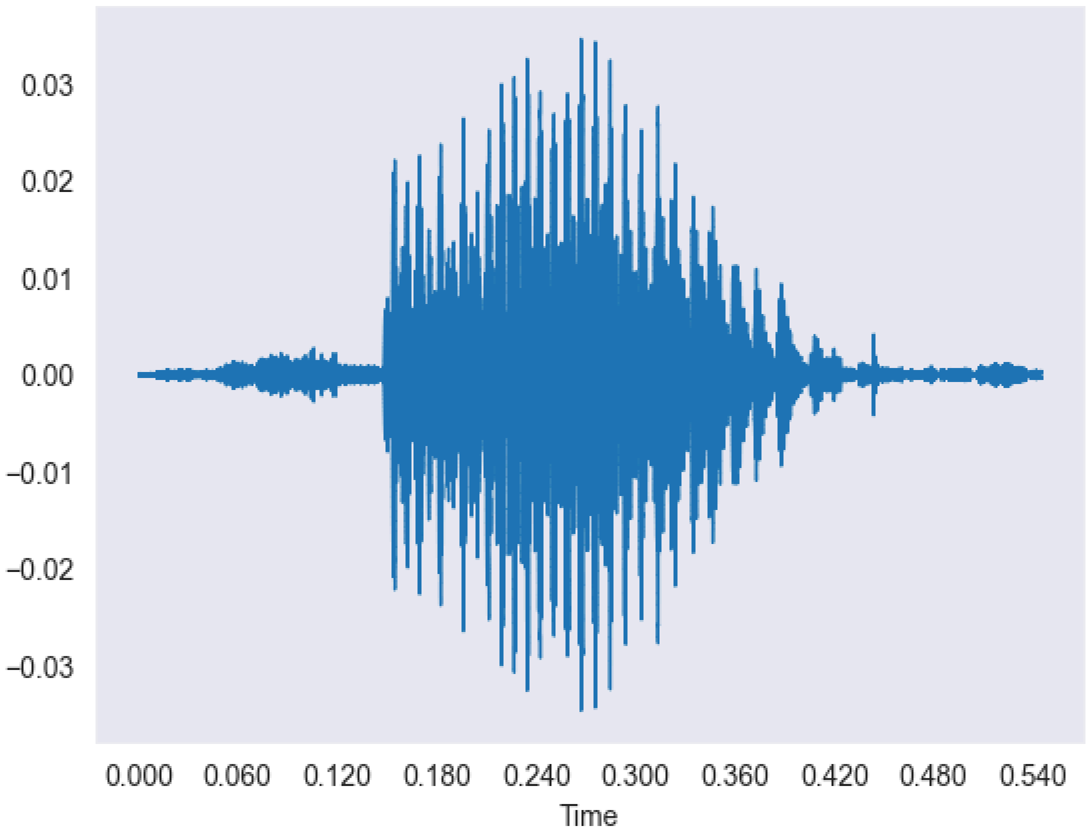


Out[7]:

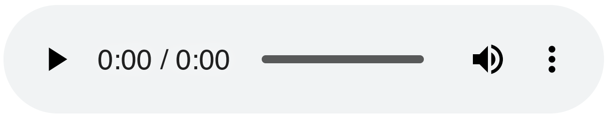


In [8]:

```
# Show the audio and plot of digit 4
get_audio(4)
```

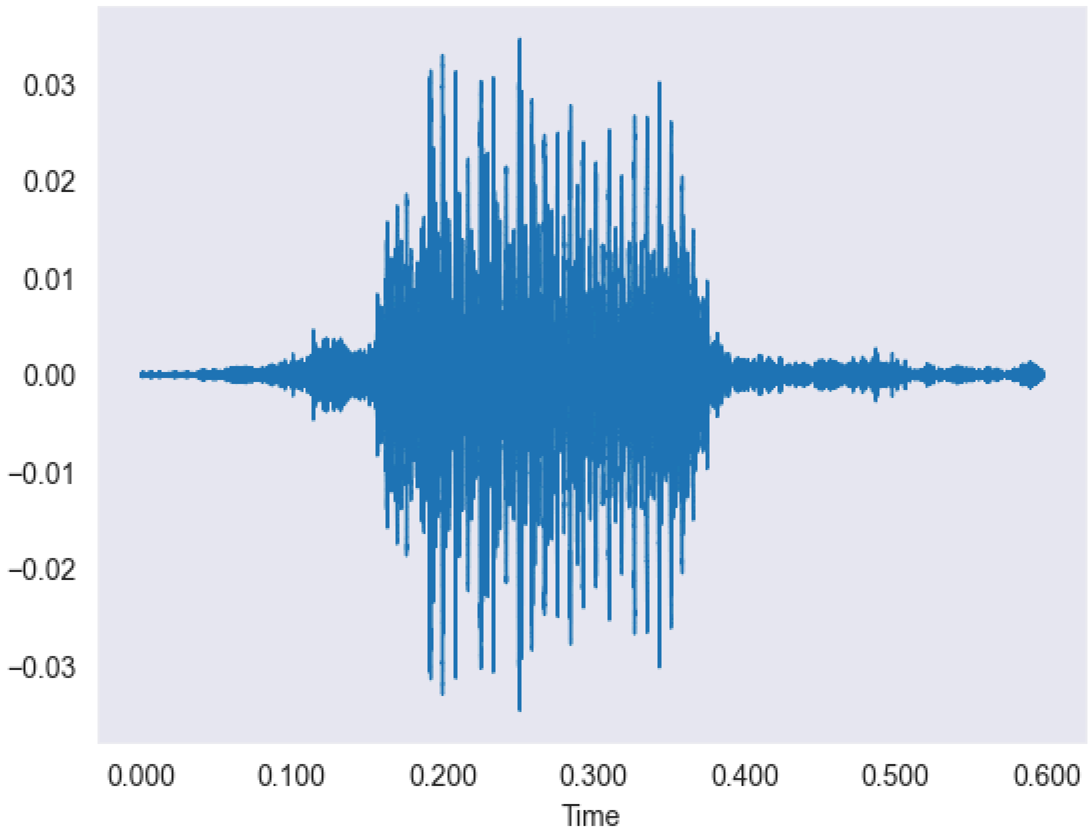


Out[8]:

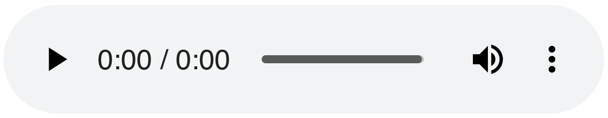


In [9]:

```
# Show the audio and plot of digit 5
get_audio(5)
```

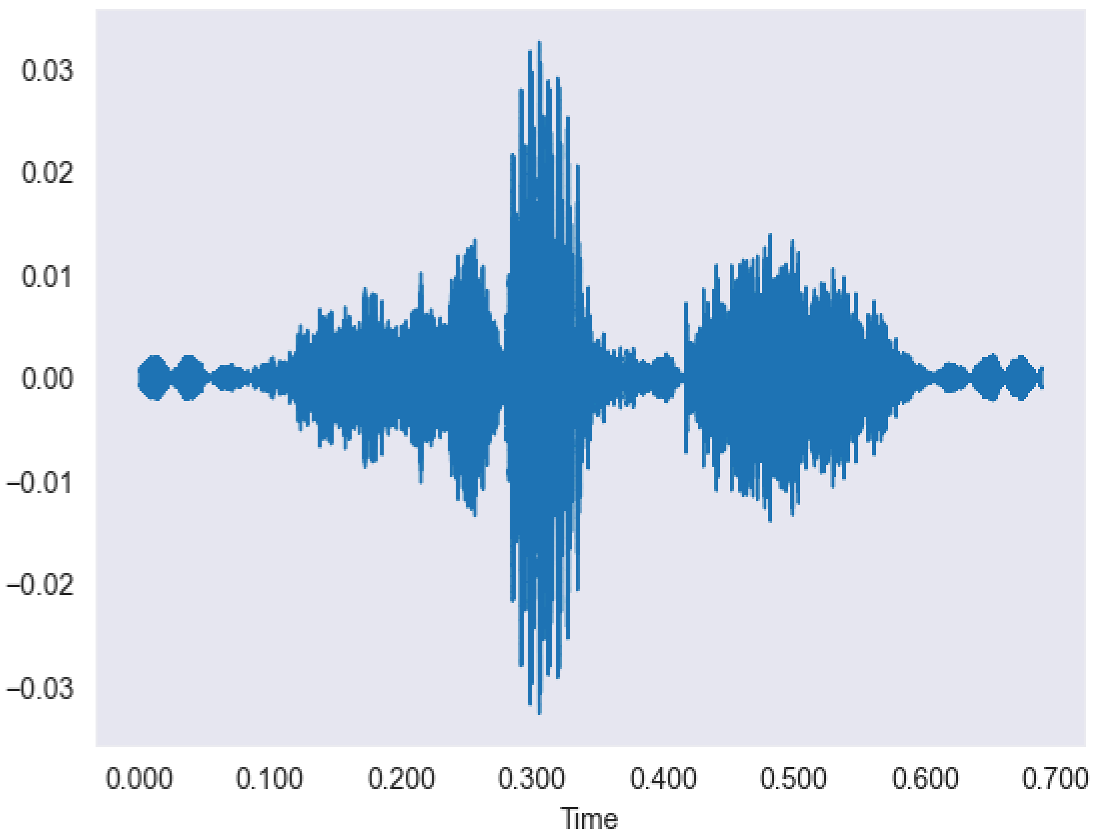


Out[9]:

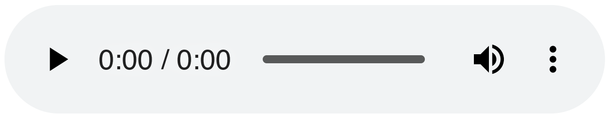


In [10]:

```
# Show the audio and plot of digit 6
get_audio(6)
```

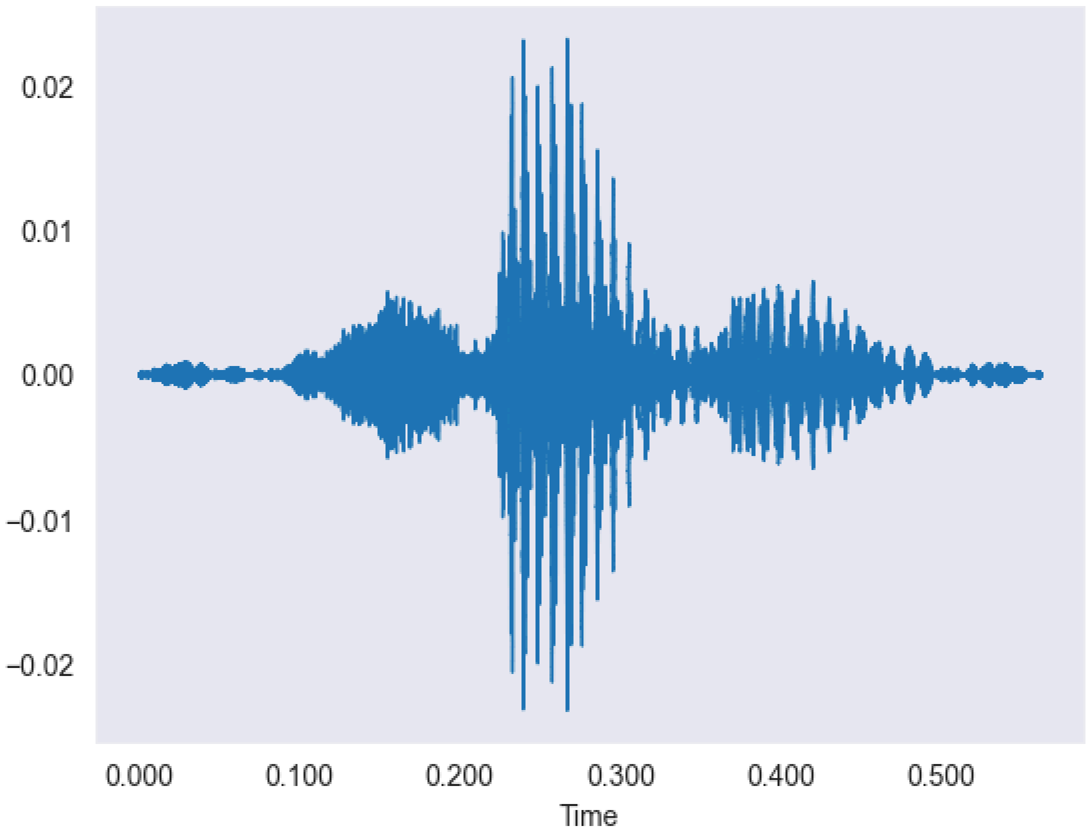


Out[10]:

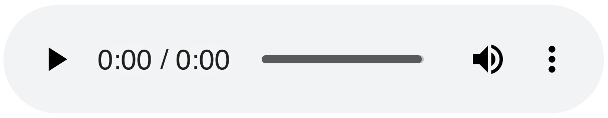


In [11]:

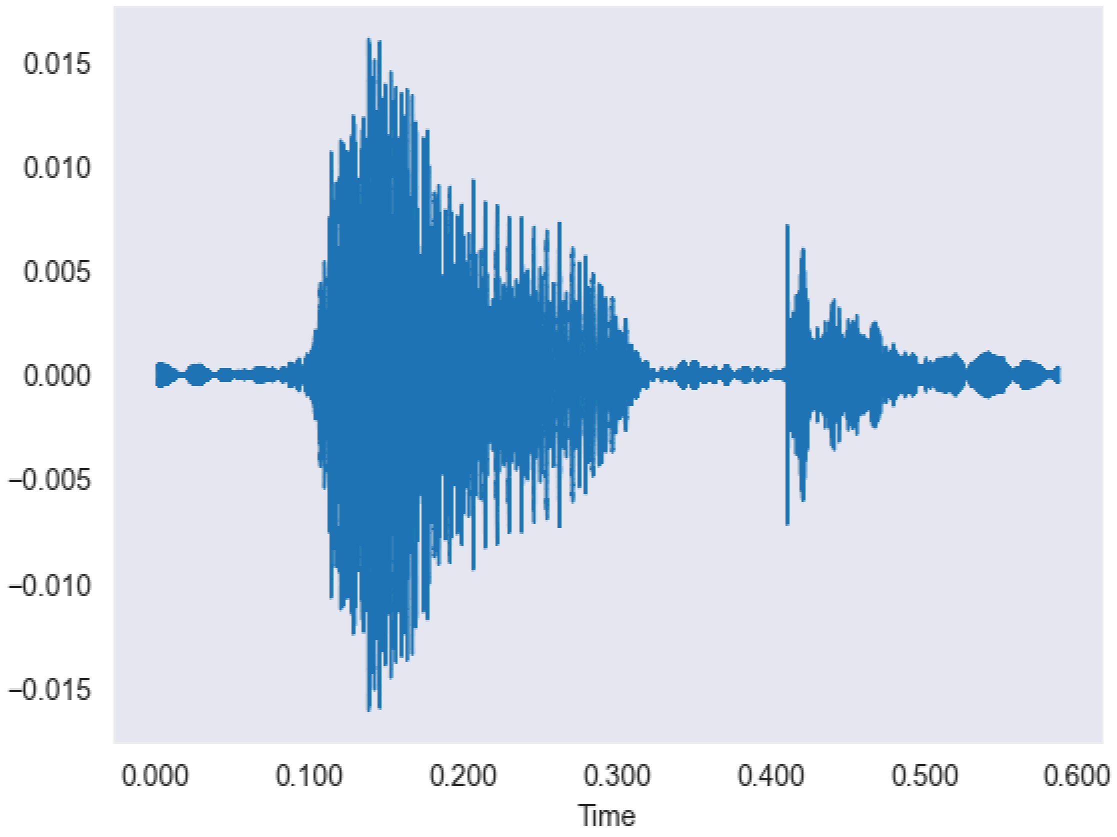
```
# Show the audio and plot of digit 7
get_audio(7)
```



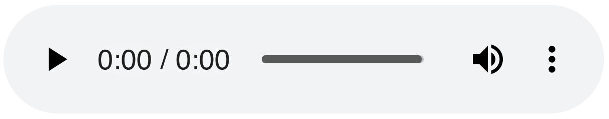
Out[11]:



In [12]:
```
# Show the audio and plot of digit 8
get_audio(8)
```

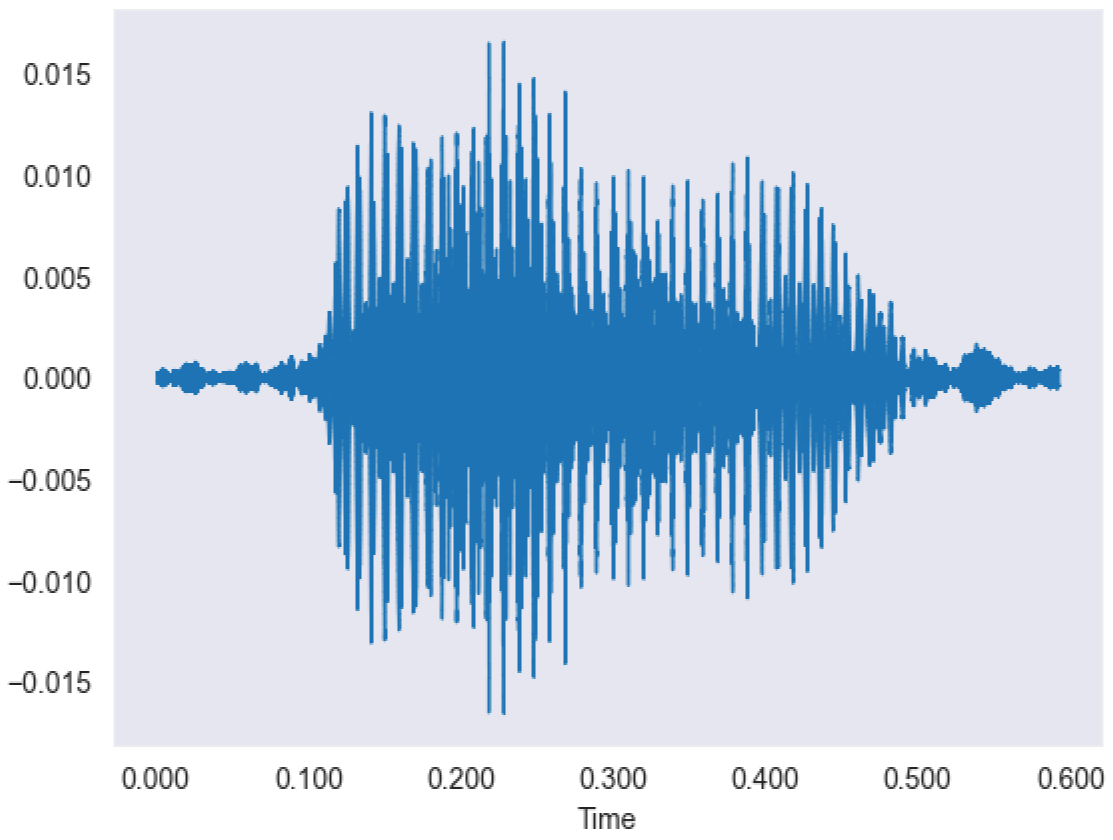


Out[12]:

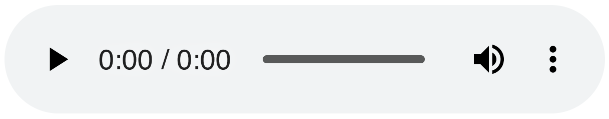


In [13]:

```
# Show the audio and plot of digit 9
get_audio(9)
```



```
Out[13]:
```



Observations:

- The X-axis represents time and Y-axis represents the amplitude of the vibrations. The intuition behind the Fourier Transform is that any wave can be broken down or deconstructed as a sum of many composite sine waves. Since these are composed of sine waves, they are symmetric about the time axis, i.e, they extend equally above and below the time axis at a particular time.
- From the various audio plots ranging from 0 to 9, we can observe the amplitude at a given point in time. For example, when we say "Zero", the "Z" sound has low amplitude and the "ero" sound has higher amplitude. Similarly, the remaining digits can be interpreted by looking at the visualizations.

Visualizing the spectrogram of the audio data

What is a spectrogram?

A spectrogram is a visual way of representing the signal strength or “loudness” of a signal over time at various frequencies or time steps present in a particular waveform. A spectrogram gives a detailed view of audio. It represents amplitude, frequency, and time in a single plot. Since spectrograms are continuous plots, they can be interpreted as an image. Different spectrograms have different attributes on their axes and they are usually different to interpret. In a Research and Development scenario, we make use of a vocoder, which is an encoder that converts spectrograms back to audio using parameters learned by machine learning. One great vocoder is the WaveNet vocoder which is used in almost all Text to Speech architectures.

Here, we will be using **MFCC** spectrograms, which are also called **Mel** spectrograms.

In [14]:
```
# A function which returns audio file for a mentioned digit
def get_audio_raw(digit = 0):

    # Audio Sample Directory
    sample = np.random.randint(1, 10)

    # Index of Audio
    index = np.random.randint(1, 5)

    # Modified file location
    if sample < 10:
        file = f"data/0{sample}/{digit}_0{sample}_{index}.wav"

    else:
        file = f"data/{sample}/{digit}_{sample}_{index}.wav"


    # Get Audio from the location
    data, sample_rate = librosa.load(file)

    # Return audio
    return data, sample_rate
```

Extracting features from the audio file

Mel-frequency cepstral coefficients (MFCCs) Feature Extraction

MFCCs are usually the final features used in many machine learning models trained on audio data. They are usually a set of mel coefficients defined for each time step through which the raw audio data can be encoded. So for example, if we have an audio sample extending for 30 time steps, and we are defining each time step by 40 Mel Coefficients, our entire sample can be represented by 40 * 30 Mel Coefficients. And if we want to create a Mel Spectrogram out of it, our spectrogram will resemble a 2-D array of 40 horizontal rows and 30 vertical columns.

In this time step, we will first extract the Mel Coefficents for each audio file and add them to our dataset.

- `extract_features` : Returns the MFCC extracted features for an audio file.

- `process_and_create_dataset` : Iterate through the audio of each digit, extract the features using the extract_features() function, and append the data into a DataFrame.

Creating a function that extracts the data from audio files

In [15]:
```
# Will take an audio file as input and return extracted features using MEL_F
def extract_features(file):

    # Load audio and its sample rate
    audio, sample_rate = librosa.load(file)

    # Extract features using mel-frequency coefficient
    extracted_features = librosa.feature.mfcc(y = audio,
                                              sr = sample_rate,
                                              n_mfcc = 40)

    # Scale the extracted features
    extracted_features = np.mean(extracted_features.T, axis = 0)

    # Return the extracted features
    return extracted_features


def preprocess_and_create_dataset():

    # Path of the folder where the audio files are present
    root_folder_path = "data/"

    # Empty List to create dataset
    dataset = []

    # Iterating through folders where each folder has the audio of each digi
    for folder in tqdm(range(1, 11)):

        if folder < 10:

            # Path of the folder
            folder = os.path.join(root_folder_path, "0" + str(folder))

        else:
            folder = os.path.join(root_folder_path, str(folder))

        # Iterate through each file of the present folder
        for file in tqdm(os.listdir(folder)):

            # Path of the file
            abs_file_path = os.path.join(folder, file)

            # Pass path of file to the extracted_features() function to crea
            extracted_features = extract_features(abs_file_path)

            # Class of the audio, i.e., the digit it represents
            class_label = file[0]
            # Append a list where the feature represents a column and class
            dataset.append([extracted_features, class_label])

    # After iterating through all the folders, convert the list to a DataFra
    return pd.DataFrame(dataset, columns = ['features', 'class'])
```

Now. let's create the dataset using the defined function

In [16]:
```
# Create the dataset by calling the function
dataset = preprocess_and_create_dataset()
```

```
  0%|
| 0/10 [00:00<?, ?it/s]
  0%|
| 0/500 [00:00<?, ?it/s]
  8%|███████
| 39/500 [00:00<00:01, 386.48it/s]
 18%|█████████████████
| 89/500 [00:00<00:00, 448.50it/s]
 28%|██████████████████████████
| 138/500 [00:00<00:00, 464.58it/s]
 38%|████████████████████████████████████
| 190/500 [00:00<00:00, 485.29it/s]
 48%|█████████████████████████████████████████████
| 240/500 [00:00<00:00, 489.02it/s]
 59%|████████████████████████████████████████████████████████
| 293/500 [00:00<00:00, 500.50it/s]
 69%|█████████████████████████████████████████████████████████████████
| 344/500 [00:00<00:00, 500.03it/s]
 79%|███████████████████████████████████████████████████████████████████████
████                    | 396/500 [00:00<00:00, 504.11it/s]
 89%|███████████████████████████████████████████████████████████████████████
██████████████          | 447/500 [00:00<00:00, 489.15it/s]
100%|███████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 478.82it/s]
 10%|██████████
| 1/10 [00:01<00:09,  1.05s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
 10%|██████████
| 50/500 [00:00<00:00, 492.31it/s]
 20%|███████████████████
| 100/500 [00:00<00:00, 422.51it/s]
 29%|███████████████████████████
| 144/500 [00:00<00:00, 427.66it/s]
 38%|████████████████████████████████████
| 188/500 [00:00<00:00, 409.86it/s]
 47%|█████████████████████████████████████████████
| 235/500 [00:00<00:00, 429.31it/s]
 57%|██████████████████████████████████████████████████████
| 286/500 [00:00<00:00, 451.74it/s]
 67%|████████████████████████████████████████████████████████████████
| 335/500 [00:00<00:00, 461.59it/s]
 77%|███████████████████████████████████████████████████████████████████████
██                      | 384/500 [00:00<00:00, 469.04it/s]
 86%|███████████████████████████████████████████████████████████████████████
███████████             | 432/500 [00:00<00:00, 458.65it/s]
100%|███████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 437.92it/s]
 20%|████████████████████
| 2/10 [00:02<00:08,  1.11s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
 11%|██████████
| 54/500 [00:00<00:00, 539.16it/s]
 22%|█████████████████████
| 108/500 [00:00<00:00, 508.57it/s]
```

```
 32%|███████████████████████████████
| 159/500 [00:00<00:00, 507.39it/s]
 43%|█████████████████████████████████████████
| 215/500 [00:00<00:00, 524.55it/s]
 54%|████████████████████████████████████████████████████
| 271/500 [00:00<00:00, 534.75it/s]
 65%|███████████████████████████████████████████████████████████████
| 325/500 [00:00<00:00, 532.33it/s]
 76%|████████████████████████████████████████████████████████████████████████
█                       | 379/500 [00:00<00:00, 530.92it/s]
 87%|████████████████████████████████████████████████████████████████████████
████████████            | 434/500 [00:00<00:00, 535.02it/s]
100%|████████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:00<00:00, 527.43it/s]
 30%|██████████████████████████████
| 3/10 [00:03<00:07,  1.04s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
  9%|█████████
| 43/500 [00:00<00:01, 426.54it/s]
 19%|███████████████████
| 97/500 [00:00<00:00, 490.50it/s]
 30%|█████████████████████████████
| 149/500 [00:00<00:00, 502.95it/s]
 40%|███████████████████████████████████████
| 200/500 [00:00<00:00, 502.03it/s]
 50%|█████████████████████████████████████████████████
| 251/500 [00:00<00:00, 496.13it/s]
 60%|██████████████████████████████████████████████████████████
| 301/500 [00:00<00:00, 465.39it/s]
 70%|███████████████████████████████████████████████████████████████████
| 348/500 [00:00<00:00, 439.43it/s]
 79%|████████████████████████████████████████████████████████████████████████
████                    | 393/500 [00:00<00:00, 408.63it/s]
 88%|████████████████████████████████████████████████████████████████████████
█████████████           | 440/500 [00:00<00:00, 425.70it/s]
100%|████████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 443.07it/s]
 40%|████████████████████████████████████████
| 4/10 [00:04<00:06,  1.07s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
  9%|█████████
| 46/500 [00:00<00:00, 454.45it/s]
 19%|███████████████████
| 97/500 [00:00<00:00, 481.09it/s]
 29%|█████████████████████████████
| 147/500 [00:00<00:00, 488.20it/s]
 39%|██████████████████████████████████████
| 197/500 [00:00<00:00, 490.36it/s]
 50%|████████████████████████████████████████████████
| 250/500 [00:00<00:00, 504.19it/s]
 60%|██████████████████████████████████████████████████████████
| 301/500 [00:00<00:00, 476.74it/s]
 70%|███████████████████████████████████████████████████████████████████
| 349/500 [00:00<00:00, 450.43it/s]
```

```
 79%|███████████████████████████████████████████████████████████████████████
████                    | 395/500 [00:00<00:00, 427.30it/s]
 89%|███████████████████████████████████████████████████████████████████████
█████████████           | 444/500 [00:00<00:00, 444.93it/s]
100%|███████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 460.04it/s]
 50%|██████████████████████████████████████████████████
| 5/10 [00:05<00:05,  1.08s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
  7%|███████
| 37/500 [00:00<00:01, 356.08it/s]
 15%|██████████████
| 73/500 [00:00<00:01, 344.75it/s]
 22%|█████████████████████
| 108/500 [00:00<00:01, 342.20it/s]
 31%|█████████████████████████████
| 154/500 [00:00<00:00, 385.10it/s]
 41%|███████████████████████████████████████
| 204/500 [00:00<00:00, 422.99it/s]
 50%|███████████████████████████████████████████████
| 248/500 [00:00<00:00, 426.04it/s]
 59%|████████████████████████████████████████████████████████
| 297/500 [00:00<00:00, 446.16it/s]
 69%|██████████████████████████████████████████████████████████████████
| 345/500 [00:00<00:00, 454.02it/s]
 79%|███████████████████████████████████████████████████████████████████████
████                    | 393/500 [00:00<00:00, 458.69it/s]
 88%|███████████████████████████████████████████████████████████████████████
█████████████           | 440/500 [00:01<00:00, 461.60it/s]
100%|███████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 429.94it/s]
 60%|████████████████████████████████████████████████████████████
| 6/10 [00:06<00:04,  1.11s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
  9%|█████████
| 46/500 [00:00<00:00, 458.99it/s]
 18%|██████████████████
| 92/500 [00:00<00:00, 424.69it/s]
 28%|███████████████████████████
| 141/500 [00:00<00:00, 450.93it/s]
 39%|█████████████████████████████████████
| 194/500 [00:00<00:00, 479.79it/s]
 49%|███████████████████████████████████████████████
| 247/500 [00:00<00:00, 497.33it/s]
 60%|█████████████████████████████████████████████████████████
| 299/500 [00:00<00:00, 502.27it/s]
 71%|███████████████████████████████████████████████████████████████████
| 353/500 [00:00<00:00, 511.81it/s]
 81%|███████████████████████████████████████████████████████████████████████
██████                  | 407/500 [00:00<00:00, 516.29it/s]
100%|███████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 487.20it/s]
 70%|██████████████████████████████████████████████████████████████████████
| 7/10 [00:07<00:03,  1.09s/it]
```

```
  0%|
| 0/500 [00:00<?, ?it/s]
  9%|█████████
| 46/500 [00:00<00:00, 458.20it/s]
 19%|███████████████████
| 97/500 [00:00<00:00, 484.90it/s]
 29%|████████████████████████████
| 147/500 [00:00<00:00, 488.52it/s]
 40%|███████████████████████████████████████
| 201/500 [00:00<00:00, 503.84it/s]
 50%|████████████████████████████████████████████████
| 252/500 [00:00<00:00, 502.14it/s]
 61%|██████████████████████████████████████████████████████████
| 303/500 [00:00<00:00, 485.38it/s]
 70%|████████████████████████████████████████████████████████████████████
| 352/500 [00:00<00:00, 470.20it/s]
 80%|████████████████████████████████████████████████████████████████████████
█████                   | 401/500 [00:00<00:00, 474.15it/s]
100%|████████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 487.29it/s]
 80%|████████████████████████████████████████████████████████████████████████
████████                    | 8/10 [00:08<00:02,  1.07s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
 10%|█████████
| 48/500 [00:00<00:00, 468.84it/s]
 19%|██████████████████
| 95/500 [00:00<00:01, 377.54it/s]
 27%|██████████████████████████
| 137/500 [00:00<00:00, 393.44it/s]
 37%|███████████████████████████████████
| 184/500 [00:00<00:00, 421.25it/s]
 47%|█████████████████████████████████████████████
| 233/500 [00:00<00:00, 442.84it/s]
 56%|██████████████████████████████████████████████████████
| 282/500 [00:00<00:00, 454.87it/s]
 66%|███████████████████████████████████████████████████████████████
| 328/500 [00:00<00:00, 427.15it/s]
 74%|███████████████████████████████████████████████████████████████████████
| 372/500 [00:00<00:00, 420.50it/s]
 83%|████████████████████████████████████████████████████████████████████████
████████                | 415/500 [00:00<00:00, 418.57it/s]
100%|████████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 429.63it/s]
 90%|████████████████████████████████████████████████████████████████████████
██████████████████          | 9/10 [00:09<00:01,  1.11s/it]
  0%|
| 0/500 [00:00<?, ?it/s]
 10%|██████████
| 50/500 [00:00<00:00, 493.10it/s]
 20%|███████████████████
| 100/500 [00:00<00:00, 465.60it/s]
 29%|████████████████████████████
| 147/500 [00:00<00:00, 463.26it/s]
 39%|█████████████████████████████████████
| 194/500 [00:00<00:00, 385.16it/s]
```

```
 49%|██████████████████████████████████████████████
| 243/500 [00:00<00:00, 416.78it/s]
 57%|███████████████████████████████████████████████████████
| 287/500 [00:00<00:00, 404.06it/s]
 68%|█████████████████████████████████████████████████████████████████
| 338/500 [00:00<00:00, 433.58it/s]
 77%|████████████████████████████████████████████████████████████████████████
██                      | 383/500 [00:01<00:00, 288.42it/s]
 84%|████████████████████████████████████████████████████████████████████████
█████████               | 419/500 [00:01<00:00, 302.23it/s]
 91%|████████████████████████████████████████████████████████████████████████
████████████████        | 456/500 [00:01<00:00, 317.44it/s]
100%|████████████████████████████████████████████████████████████████████████
████████████████████████| 500/500 [00:01<00:00, 360.03it/s]
100%|████████████████████████████████████████████████████████████████████████
███████████████████████████| 10/10 [00:11<00:00,  1.12s/it]
```

View first 5 rows of the data

In [17]:
```python
# View the head of the DataFrame
dataset.head()
```

Out[17]:

	features	class
0	[-634.2726, 60.718662, 19.634466, 54.73299, 31...	3
1	[-615.57446, 117.79709, 20.75868, 23.231092, 2...	2
2	[-632.2322, 63.031258, 15.4895, 50.909485, 32....	3
3	[-619.4891, 70.50161, 7.5741982, 47.71054, 36....	3
4	[-611.01135, 110.0762, 19.848354, 13.996176, 8...	4

In [18]:
```python
dataset.shape
```

Out[18]:
```
(5000, 2)
```

In [19]:
```python
dataset.describe().T
```

Out[19]:

	count	unique	top	freq
features	5000	5000	[-634.2726, 60.718662, 19.634466, 54.73299, 31...	1
class	5000	10	3	500

In [20]:
```python
# Storing the class as int
dataset['class'] = [int(x) for x in dataset['class']]
```

In [21]:
```python
# Check the frequency of classes in the dataset
dataset['class'].value_counts()
```

Out[21]:
```
class
3    500
2    500
4    500
0    500
7    500
8    500
5    500
1    500
6    500
9    500
Name: count, dtype: int64
```

Visualizing the Mel Frequency Cepstral Coefficients Using a Spectrogram

- `draw_spectrograms` : From the Mel Coefficients we are extracting for a particular audio, this function is creating the 2-D graph of those coefficients with the X-axis representing time and the Y-axis shows the corresponding Mel coefficients in that time step.

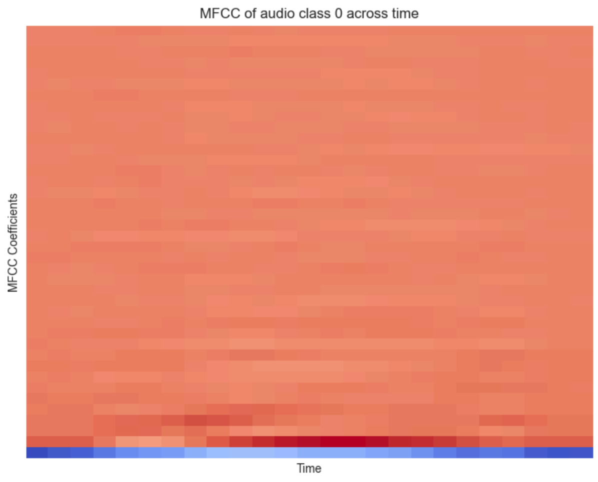
In [22]:
```python
# A function which returns MFCC
def draw_spectrograms(audio_data, sample_rate):

    # Extract features
    extracted_features = librosa.feature.mfcc(y = audio_data,
                                              sr = sample_rate,
                                              n_mfcc = 40)

    # Return features without scaling
    return extracted_features
```

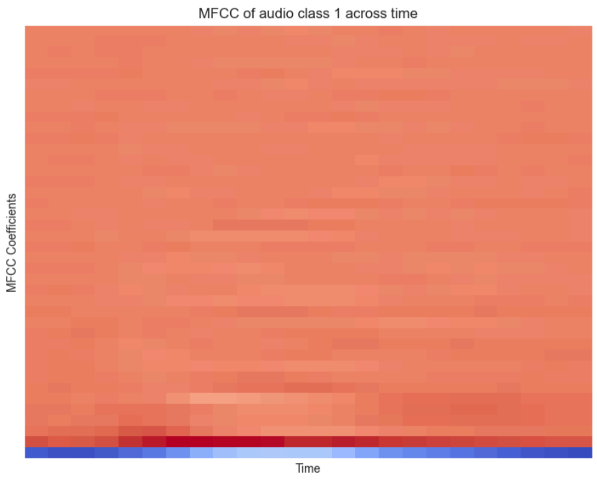
The very first MFCC coefficient (0th coefficient) does not provide information about the overall shape of the spectrum. It simply communicates a constant offset or the addition of a constant value to the full spectrum. As a result, when performing classification, many practitioners will disregard the initial MFCC. In the images, you can see those represented by blue pixels.

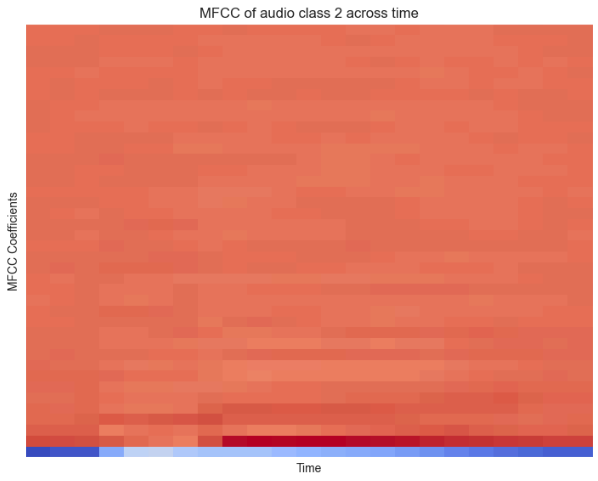
We can plot the MFCCs, but it's difficult to tell what kind of signal is hiding behind such representation.

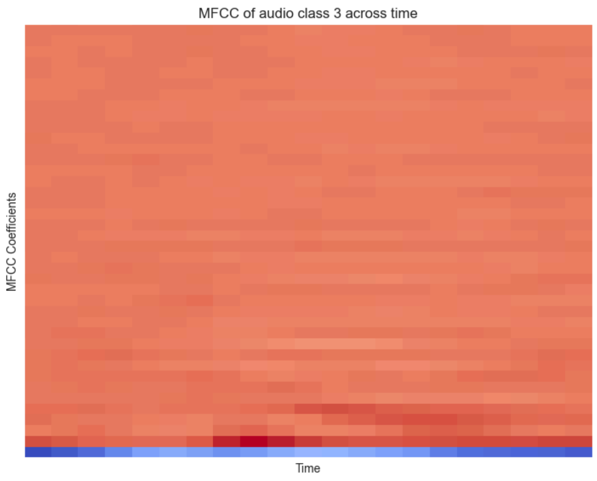
In [23]:
```python
# Creating subplots
fig, ax = plt.subplots(5, 2, figsize = (15, 30))

# Initializing row and column variables for subplots
row = 0
column = 0

for digit in range(10):

    # Get the audio of different classes (0-9)
    audio_data, sample_rate = get_audio_raw(digit)

    # Extract their MFCC
    mfcc = draw_spectrograms(audio_data, sample_rate)
    print(f"Shape of MFCC of audio digit {digit} ---> ", mfcc.shape)

    # Display the plots and its title
    ax[row,column].set_title(f"MFCC of audio class {digit} across time")
    librosa.display.specshow(mfcc, sr = 22050, ax = ax[row, column])

    # Set X-labels and Y-labels
    ax[row,column].set_xlabel("Time")
    ax[row,column].set_ylabel("MFCC Coefficients")

    # Conditions for positioning of the plots
    if column == 1:
        column = 0
        row += 1
    else:
        column+=1


plt.tight_layout(pad = 3)
plt.show()
```

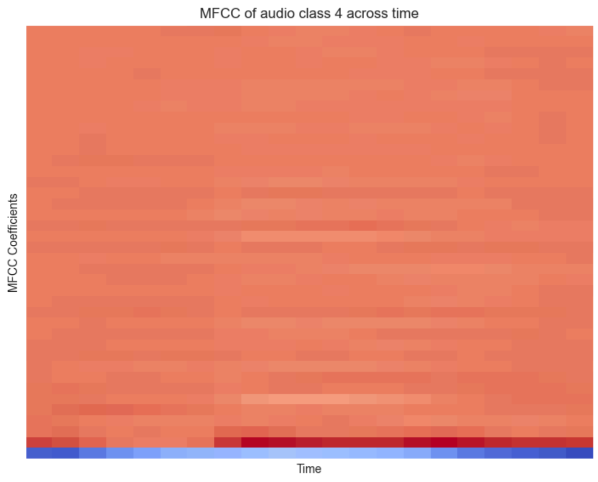
```
Shape of MFCC of audio digit 0 --->  (40, 25)
Shape of MFCC of audio digit 1 --->  (40, 24)
Shape of MFCC of audio digit 2 --->  (40, 23)
Shape of MFCC of audio digit 3 --->  (40, 21)
Shape of MFCC of audio digit 4 --->  (40, 21)
Shape of MFCC of audio digit 5 --->  (40, 22)
Shape of MFCC of audio digit 6 --->  (40, 40)
Shape of MFCC of audio digit 7 --->  (40, 26)
Shape of MFCC of audio digit 8 --->  (40, 23)
Shape of MFCC of audio digit 9 --->  (40, 24)
```

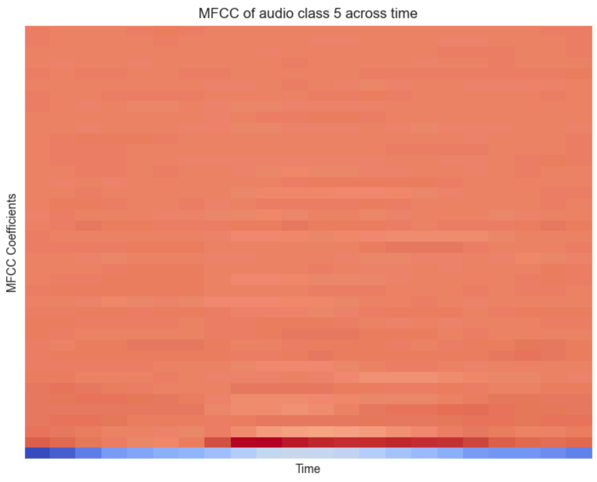

MFCC of audio class 0 across time
MFCC Coefficients
Time

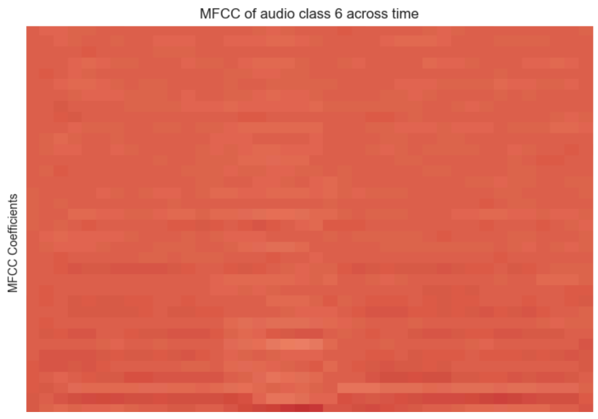

MFCC of audio class 1 across time
MFCC Coefficients
Time

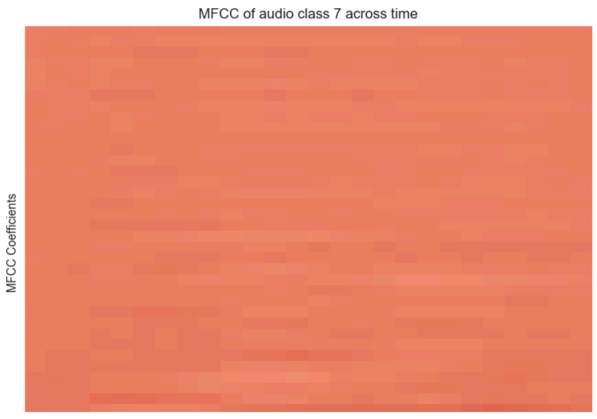

MFCC of audio class 2 across time
MFCC Coefficients
Time

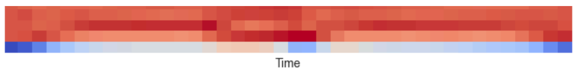

MFCC of audio class 3 across time
MFCC Coefficients
Time

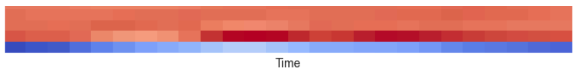

MFCC of audio class 4 across time
MFCC Coefficients
Time

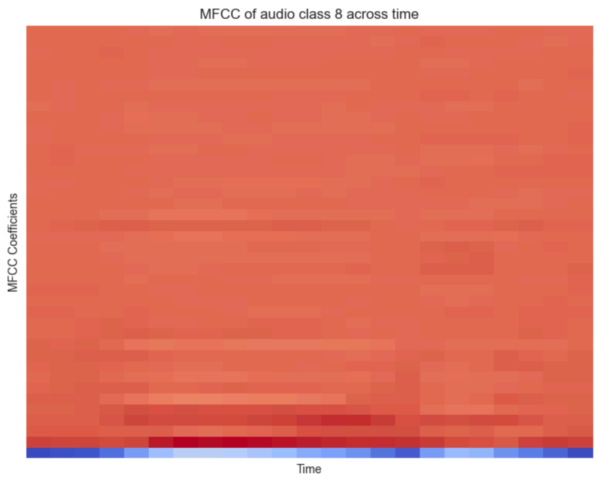

MFCC of audio class 5 across time
MFCC Coefficients
Time

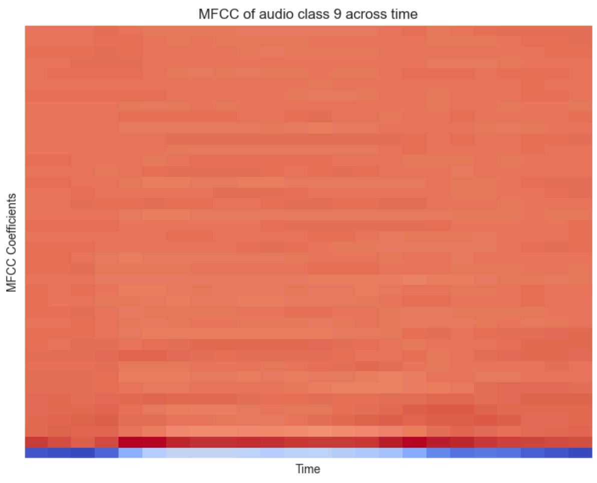

MFCC of audio class 6 across time
MFCC Coefficients


MFCC of audio class 7 across time
MFCC Coefficients









Visual Inspection of MFCC Spectrograms:

On inspecting them visually, we can see that there are a lot of deviations from the spectrograms of one audio to another. There are a lot of tiny rectangles and bars whose positions are unique to each audio. So, the Artificial Neural Network should be able to perform decently in identifying these audios.

Perform Train-Test-Split

- Split the data into train and test sets

In [24]:
```
# Import train_test_split function
from sklearn.model_selection import train_test_split

X = np.array(dataset['features'].to_list())
Y = np.array(dataset['class'].to_list())

# Create train set and test set
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, train_size = 0.75,
```

In [25]:
```
# Checking the shape of the data
X_train.shape
```

Out[25]:
```
(3750, 40)
```

Modelling

- Create an artificial neural network to recognize the digit.

About the libraries:

- `Keras` : Keras is an open-source deep-learning library in Python. Keras is popular because the API was clean and simple, allowing standard deep learning models to be defined, fit, and evaluated in just a few lines of code.
- `Sklearn` :
    - Simple and efficient tools for predictive data analysis
    - Accessible to everybody, and reusable in various contexts
    - Built on NumPy, SciPy, and matplotlib
    - Open source, commercially usable

Import necessary libraries for building the model

In [26]:

```
# To create an ANN model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# To create a checkpoint and save the best model
from tensorflow.keras.callbacks import ModelCheckpoint

# To load the model
from tensorflow.keras.models import load_model

# To evaluate the model
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.preprocessing import LabelBinarizer
```

Model Creation

Why are we using ANN's?

When we are converting audios to their corresponding spectrograms, we will have similar spectrograms for similar audios irrespective of who the speaker is, and what is their pitch and timber like. So local spatiality is never going to be a problem. So having convolutional layers on top of our fully connected layers is just adding to our computational redundancy.

We will use a Sequential model with multiple connected hidden layers, and an output layer that returns a single, continuous value.

- A Sequential model is a linear stack of layers. Sequential models can be created by giving a list of layer instances.
- A dense layer of neurons is a simple layer of neurons in which each neuron receives input from all of the neurons in the previous layer.
- The most popular function employed for hidden layers is the rectified linear activation function, or ReLU activation function. It's popular because it's easy to use and effective in getting around the limitations of other popular activation functions like Sigmoid and Tanh.

In [27]:
```
# Crete a Sequential Object
model = Sequential()

# Add first layer with 100 neurons to the sequental object
model.add(Dense(100, input_shape = (40, ), activation = 'relu'))

# Add second layer with 100 neurons to the sequental object
model.add(Dense(100, activation = 'relu'))

# Add third layer with 100 neurons to the sequental object
model.add(Dense(100, activation = 'relu'))

# Output layer with 10 neurons as it has 10 classes
model.add(Dense(10, activation = 'softmax'))
```

```
/Users/obaozai/miniconda3/envs/jupyter/lib/python3.11/site-packages/keras/sr
c/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_
dim` argument to a layer. When using Sequential models, prefer using an `Inp
ut(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [28]:
```
# Print Summary of the model
model.summary()
```

Model: "sequential"

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━
┃ Layer (type)                      ┃ Output Shape           ┃      Par
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━
│ dense (Dense)                     │ (None, 100)            │        4
├───────────────────────────────────┼────────────────────────┼─────────
│ dense_1 (Dense)                   │ (None, 100)            │       10
├───────────────────────────────────┼────────────────────────┼─────────
│ dense_2 (Dense)                   │ (None, 100)            │       10
├───────────────────────────────────┼────────────────────────┼─────────
│ dense_3 (Dense)                   │ (None, 10)             │        1
└───────────────────────────────────┴────────────────────────┴─────────
 Total params: 25,310 (98.87 KB)
 Trainable params: 25,310 (98.87 KB)
 Non-trainable params: 0 (0.00 B)
```

In [29]:
```python
# Compile the model
model.compile(loss = 'sparse_categorical_crossentropy',
              metrics = ['accuracy'],
              optimizer = 'adam')
```

Model Checkpoint & Training

In [30]:
```python
# Set the number of epochs for training
num_epochs = 100

# Set the batch size for training
batch_size = 32

# Fit the model
model.fit(X_train, Y_train, validation_data = (X_test, Y_test), epochs = num
Epoch 1/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.2667 - loss: 7.4935 -
val_accuracy: 0.7568 - val_loss: 0.7217
Epoch 2/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 635us/step - accuracy: 0.8055 - loss: 0.5902
- val_accuracy: 0.7648 - val_loss: 0.5980
Epoch 3/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 622us/step - accuracy: 0.8566 - loss: 0.4161
- val_accuracy: 0.8888 - val_loss: 0.3074
Epoch 4/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 614us/step - accuracy: 0.9175 - loss: 0.2538
- val_accuracy: 0.9192 - val_loss: 0.2091
Epoch 5/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 608us/step - accuracy: 0.9293 - loss: 0.2207
- val_accuracy: 0.9328 - val_loss: 0.1806
Epoch 6/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 617us/step - accuracy: 0.9426 - loss: 0.1679
- val_accuracy: 0.9744 - val_loss: 0.0909
Epoch 7/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 613us/step - accuracy: 0.9602 - loss: 0.1277
- val_accuracy: 0.9736 - val_loss: 0.0909
Epoch 8/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 632us/step - accuracy: 0.9538 - loss: 0.1300
- val_accuracy: 0.9536 - val_loss: 0.1094
Epoch 9/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - accuracy: 0.9761 - loss: 0.0727
- val_accuracy: 0.9728 - val_loss: 0.0870
Epoch 10/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - accuracy: 0.9660 - loss: 0.1024
- val_accuracy: 0.9512 - val_loss: 0.1381
Epoch 11/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 643us/step - accuracy: 0.9610 - loss: 0.1195
- val_accuracy: 0.9568 - val_loss: 0.1174
Epoch 12/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 662us/step - accuracy: 0.9666 - loss: 0.0967
- val_accuracy: 0.9760 - val_loss: 0.0782
Epoch 13/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 670us/step - accuracy: 0.9760 - loss: 0.0724
- val_accuracy: 0.9776 - val_loss: 0.0634
Epoch 14/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 628us/step - accuracy: 0.9755 - loss: 0.0676
- val_accuracy: 0.9760 - val_loss: 0.0663
Epoch 15/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - accuracy: 0.9781 - loss: 0.0695
- val_accuracy: 0.9856 - val_loss: 0.0445
Epoch 16/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 617us/step - accuracy: 0.9821 - loss: 0.0448
- val_accuracy: 0.9704 - val_loss: 0.0880
Epoch 17/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - accuracy: 0.9630 - loss: 0.0904
- val_accuracy: 0.9784 - val_loss: 0.0765
Epoch 18/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 658us/step - accuracy: 0.9617 - loss: 0.1092
- val_accuracy: 0.9672 - val_loss: 0.0964
Epoch 19/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 652us/step - accuracy: 0.9672 - loss: 0.0845
```

```
- val_accuracy: 0.9688 - val_loss: 0.0750
Epoch 20/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - accuracy: 0.9802 - loss: 0.0590
- val_accuracy: 0.9696 - val_loss: 0.0861
Epoch 21/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 730us/step - accuracy: 0.9813 - loss: 0.0571
- val_accuracy: 0.9808 - val_loss: 0.0583
Epoch 22/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 670us/step - accuracy: 0.9707 - loss: 0.0819
- val_accuracy: 0.9888 - val_loss: 0.0402
Epoch 23/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 653us/step - accuracy: 0.9846 - loss: 0.0439
- val_accuracy: 0.9576 - val_loss: 0.1189
Epoch 24/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 646us/step - accuracy: 0.9799 - loss: 0.0588
- val_accuracy: 0.9600 - val_loss: 0.1199
Epoch 25/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 657us/step - accuracy: 0.9844 - loss: 0.0447
- val_accuracy: 0.9904 - val_loss: 0.0314
Epoch 26/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 678us/step - accuracy: 0.9818 - loss: 0.0493
- val_accuracy: 0.9768 - val_loss: 0.0687
Epoch 27/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 659us/step - accuracy: 0.9714 - loss: 0.0733
- val_accuracy: 0.9728 - val_loss: 0.0709
Epoch 28/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 663us/step - accuracy: 0.9763 - loss: 0.0817
- val_accuracy: 0.9720 - val_loss: 0.0994
Epoch 29/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 653us/step - accuracy: 0.9810 - loss: 0.0548
- val_accuracy: 0.9312 - val_loss: 0.2456
Epoch 30/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 658us/step - accuracy: 0.9845 - loss: 0.0548
- val_accuracy: 0.9680 - val_loss: 0.0693
Epoch 31/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 647us/step - accuracy: 0.9853 - loss: 0.0378
- val_accuracy: 0.9824 - val_loss: 0.0601
Epoch 32/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 654us/step - accuracy: 0.9821 - loss: 0.0522
- val_accuracy: 0.9600 - val_loss: 0.1364
Epoch 33/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 615us/step - accuracy: 0.9779 - loss: 0.0727
- val_accuracy: 0.9840 - val_loss: 0.0502
Epoch 34/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 650us/step - accuracy: 0.9785 - loss: 0.0729
- val_accuracy: 0.9680 - val_loss: 0.0920
Epoch 35/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 634us/step - accuracy: 0.9753 - loss: 0.0798
- val_accuracy: 0.9880 - val_loss: 0.0354
Epoch 36/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 653us/step - accuracy: 0.9927 - loss: 0.0204
- val_accuracy: 0.9808 - val_loss: 0.0490
Epoch 37/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 650us/step - accuracy: 0.9873 - loss: 0.0415
- val_accuracy: 0.9224 - val_loss: 0.2533
Epoch 38/100
```

```
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 625us/step - accuracy: 0.9801 - loss: 0.0633
- val_accuracy: 0.9744 - val_loss: 0.0759
Epoch 39/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 640us/step - accuracy: 0.9803 - loss: 0.0507
- val_accuracy: 0.9848 - val_loss: 0.0512
Epoch 40/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 622us/step - accuracy: 0.9826 - loss: 0.0510
- val_accuracy: 0.9824 - val_loss: 0.0548
Epoch 41/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 663us/step - accuracy: 0.9904 - loss: 0.0286
- val_accuracy: 0.9824 - val_loss: 0.0596
Epoch 42/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 656us/step - accuracy: 0.9812 - loss: 0.0507
- val_accuracy: 0.9880 - val_loss: 0.0545
Epoch 43/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 619us/step - accuracy: 0.9912 - loss: 0.0305
- val_accuracy: 0.9832 - val_loss: 0.0566
Epoch 44/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 645us/step - accuracy: 0.9928 - loss: 0.0198
- val_accuracy: 0.9800 - val_loss: 0.0604
Epoch 45/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 636us/step - accuracy: 0.9753 - loss: 0.1101
- val_accuracy: 0.9840 - val_loss: 0.0454
Epoch 46/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 632us/step - accuracy: 0.9889 - loss: 0.0245
- val_accuracy: 0.9808 - val_loss: 0.0650
Epoch 47/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 618us/step - accuracy: 0.9939 - loss: 0.0156
- val_accuracy: 0.9784 - val_loss: 0.0629
Epoch 48/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - accuracy: 0.9915 - loss: 0.0196
- val_accuracy: 0.9848 - val_loss: 0.0464
Epoch 49/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 660us/step - accuracy: 0.9898 - loss: 0.0294
- val_accuracy: 0.9880 - val_loss: 0.0431
Epoch 50/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 647us/step - accuracy: 0.9867 - loss: 0.0357
- val_accuracy: 0.9760 - val_loss: 0.0812
Epoch 51/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 631us/step - accuracy: 0.9912 - loss: 0.0253
- val_accuracy: 0.9872 - val_loss: 0.0442
Epoch 52/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 642us/step - accuracy: 0.9949 - loss: 0.0138
- val_accuracy: 0.9864 - val_loss: 0.0346
Epoch 53/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 655us/step - accuracy: 0.9938 - loss: 0.0177
- val_accuracy: 0.9840 - val_loss: 0.0516
Epoch 54/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 637us/step - accuracy: 0.9890 - loss: 0.0252
- val_accuracy: 0.9912 - val_loss: 0.0277
Epoch 55/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - accuracy: 0.9921 - loss: 0.0239
- val_accuracy: 0.9832 - val_loss: 0.0671
Epoch 56/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - accuracy: 0.9857 - loss: 0.0349
- val_accuracy: 0.9888 - val_loss: 0.0358
```

```
Epoch 57/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 659us/step - accuracy: 0.9883 - loss: 0.0442
- val_accuracy: 0.9848 - val_loss: 0.0539
Epoch 58/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - accuracy: 0.9903 - loss: 0.0220
- val_accuracy: 0.9864 - val_loss: 0.0517
Epoch 59/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 645us/step - accuracy: 0.9986 - loss: 0.0066
- val_accuracy: 0.9880 - val_loss: 0.0529
Epoch 60/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 666us/step - accuracy: 0.9913 - loss: 0.0349
- val_accuracy: 0.9864 - val_loss: 0.0355
Epoch 61/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 647us/step - accuracy: 0.9885 - loss: 0.0391
- val_accuracy: 0.9888 - val_loss: 0.0329
Epoch 62/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 628us/step - accuracy: 0.9972 - loss: 0.0056
- val_accuracy: 0.9928 - val_loss: 0.0278
Epoch 63/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - accuracy: 0.9985 - loss: 0.0057
- val_accuracy: 0.9904 - val_loss: 0.0352
Epoch 64/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9927 - loss: 0.0181 -
val_accuracy: 0.9848 - val_loss: 0.0600
Epoch 65/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - accuracy: 0.9921 - loss: 0.0228
- val_accuracy: 0.9896 - val_loss: 0.0352
Epoch 66/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 659us/step - accuracy: 0.9988 - loss: 0.0040
- val_accuracy: 0.9720 - val_loss: 0.0952
Epoch 67/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 618us/step - accuracy: 0.9913 - loss: 0.0272
- val_accuracy: 0.9848 - val_loss: 0.0545
Epoch 68/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 613us/step - accuracy: 0.9968 - loss: 0.0085
- val_accuracy: 0.9912 - val_loss: 0.0274
Epoch 69/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 625us/step - accuracy: 0.9963 - loss: 0.0119
- val_accuracy: 0.9872 - val_loss: 0.0626
Epoch 70/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 615us/step - accuracy: 0.9908 - loss: 0.0244
- val_accuracy: 0.9928 - val_loss: 0.0291
Epoch 71/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 629us/step - accuracy: 0.9947 - loss: 0.0121
- val_accuracy: 0.9920 - val_loss: 0.0261
Epoch 72/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 629us/step - accuracy: 1.0000 - loss: 6.0500
e-04 - val_accuracy: 0.9936 - val_loss: 0.0246
Epoch 73/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 614us/step - accuracy: 1.0000 - loss: 2.9582
e-04 - val_accuracy: 0.9928 - val_loss: 0.0259
Epoch 74/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 628us/step - accuracy: 1.0000 - loss: 1.7942
e-04 - val_accuracy: 0.9952 - val_loss: 0.0247
Epoch 75/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 640us/step - accuracy: 1.0000 - loss: 1.3299
```

```
e-04 - val_accuracy: 0.9936 - val_loss: 0.0259
Epoch 76/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 614us/step - accuracy: 1.0000 - loss: 1.7931
e-04 - val_accuracy: 0.9928 - val_loss: 0.0261
Epoch 77/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - accuracy: 1.0000 - loss: 1.7376
e-04 - val_accuracy: 0.9928 - val_loss: 0.0256
Epoch 78/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 611us/step - accuracy: 1.0000 - loss: 1.4021
e-04 - val_accuracy: 0.9928 - val_loss: 0.0265
Epoch 79/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - accuracy: 1.0000 - loss: 1.1809
e-04 - val_accuracy: 0.9928 - val_loss: 0.0257
Epoch 80/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - accuracy: 1.0000 - loss: 1.1499
e-04 - val_accuracy: 0.9928 - val_loss: 0.0258
Epoch 81/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 630us/step - accuracy: 1.0000 - loss: 1.2501
e-04 - val_accuracy: 0.9928 - val_loss: 0.0254
Epoch 82/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 631us/step - accuracy: 1.0000 - loss: 9.3003
e-05 - val_accuracy: 0.9928 - val_loss: 0.0266
Epoch 83/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 623us/step - accuracy: 1.0000 - loss: 1.0386
e-04 - val_accuracy: 0.9936 - val_loss: 0.0261
Epoch 84/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - accuracy: 1.0000 - loss: 8.9002
e-05 - val_accuracy: 0.9928 - val_loss: 0.0256
Epoch 85/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 620us/step - accuracy: 1.0000 - loss: 8.4360
e-05 - val_accuracy: 0.9936 - val_loss: 0.0268
Epoch 86/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 611us/step - accuracy: 1.0000 - loss: 8.6205
e-05 - val_accuracy: 0.9936 - val_loss: 0.0264
Epoch 87/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 619us/step - accuracy: 1.0000 - loss: 5.4788
e-05 - val_accuracy: 0.9944 - val_loss: 0.0261
Epoch 88/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 619us/step - accuracy: 1.0000 - loss: 6.8237
e-05 - val_accuracy: 0.9920 - val_loss: 0.0262
Epoch 89/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 611us/step - accuracy: 1.0000 - loss: 6.9803
e-05 - val_accuracy: 0.9928 - val_loss: 0.0262
Epoch 90/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 629us/step - accuracy: 1.0000 - loss: 7.5340
e-05 - val_accuracy: 0.9928 - val_loss: 0.0275
Epoch 91/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 623us/step - accuracy: 1.0000 - loss: 6.2406
e-05 - val_accuracy: 0.9928 - val_loss: 0.0270
Epoch 92/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - accuracy: 1.0000 - loss: 5.2389
e-05 - val_accuracy: 0.9928 - val_loss: 0.0269
Epoch 93/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 623us/step - accuracy: 1.0000 - loss: 5.1984
e-05 - val_accuracy: 0.9936 - val_loss: 0.0269
Epoch 94/100
```

```
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 609us/step - accuracy: 1.0000 - loss: 5.4045
e-05 - val_accuracy: 0.9936 - val_loss: 0.0274
Epoch 95/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 620us/step - accuracy: 1.0000 - loss: 5.4869
e-05 - val_accuracy: 0.9936 - val_loss: 0.0281
Epoch 96/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 616us/step - accuracy: 1.0000 - loss: 5.9503
e-05 - val_accuracy: 0.9936 - val_loss: 0.0254
Epoch 97/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 615us/step - accuracy: 1.0000 - loss: 6.1865
e-05 - val_accuracy: 0.9936 - val_loss: 0.0272
Epoch 98/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - accuracy: 1.0000 - loss: 4.4272
e-05 - val_accuracy: 0.9936 - val_loss: 0.0267
Epoch 99/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 628us/step - accuracy: 1.0000 - loss: 2.9901
e-05 - val_accuracy: 0.9936 - val_loss: 0.0262
Epoch 100/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 627us/step - accuracy: 1.0000 - loss: 6.8726
e-05 - val_accuracy: 0.9944 - val_loss: 0.0271
```

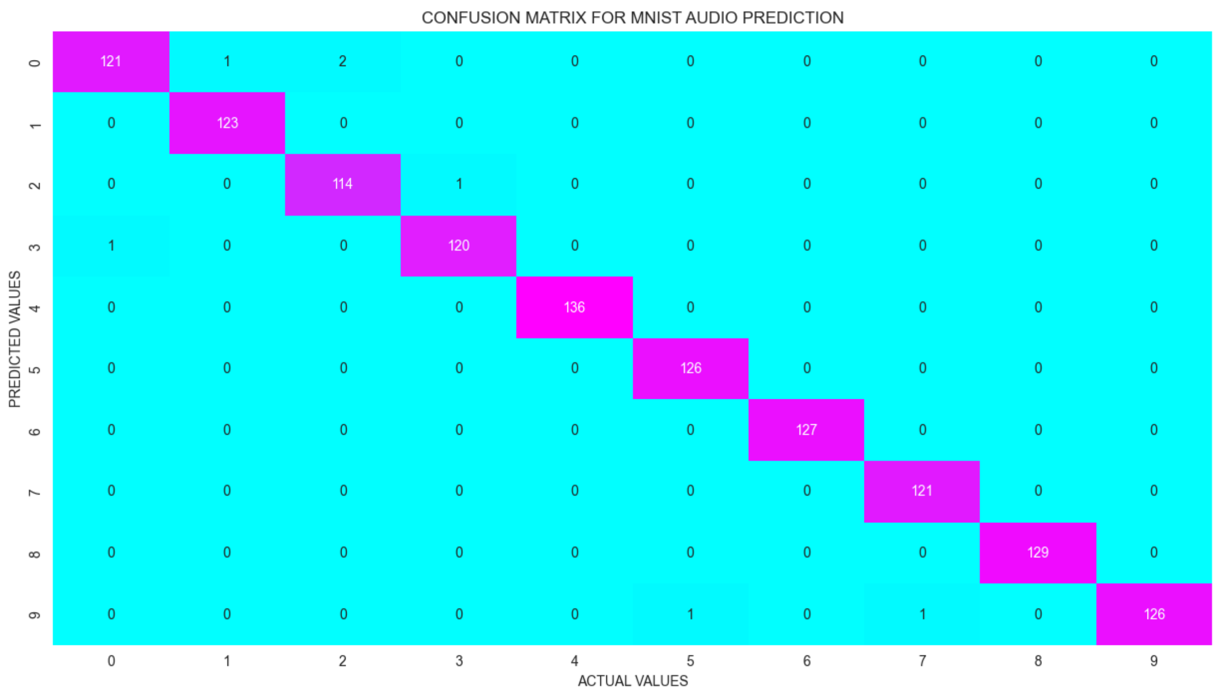
```
Out[30]: <keras.src.callbacks.history.History at 0x35487e5d0>
```

Model Evaluation

```
In [31]: # Make predictions on the test set
Y_pred = model.predict(X_test)
Y_pred = [np.argmax(i) for i in Y_pred]
```

```
40/40 ━━━━━━━━━━━━━━━━━━━━ 0s 566us/step
```

```
In [32]: # Set style as dark
sns.set_style("dark")

# Set figure size
plt.figure(figsize = (15, 8))

# Plot the title
plt.title("CONFUSION MATRIX FOR MNIST AUDIO PREDICTION")

# Confusion matrix
cm = confusion_matrix([int(x) for x in Y_test], Y_pred)

# Plot the confusion matrix as heatmap
sns.heatmap(cm, annot = True, cmap = "cool", fmt = 'g', cbar = False)

# Set X-label and Y-label
plt.xlabel("ACTUAL VALUES")
plt.ylabel("PREDICTED VALUES")

# Show the plot
plt.show()

# Print the metrics
print(classification_report(Y_test, Y_pred))
```



```
              precision    recall  f1-score   support

           0       0.99      0.98      0.98       124
           1       0.99      1.00      1.00       123
           2       0.98      0.99      0.99       115
           3       0.99      0.99      0.99       121
           4       1.00      1.00      1.00       136
           5       0.99      1.00      1.00       126
           6       1.00      1.00      1.00       127
           7       0.99      1.00      1.00       121
           8       1.00      1.00      1.00       129
           9       1.00      0.98      0.99       128

    accuracy                           0.99      1250
   macro avg       0.99      0.99      0.99      1250
weighted avg       0.99      0.99      0.99      1250
```

Observations:

- From the confusion matrix, we can observe that most of the observations are correctly identified by the model.
- In very few cases, the model is not able to identify the correct digit. For example, 9 observations are 0 but the model has predicted them as 2.
- The model has given a great performance with 99% recall, precision and F1-score.