Detecting And Localizing People Under Rubbles

Spring - 2022/2023

MKT2812 – Signals and Systems

Project Report

Group Members: Göktuğ Can Şimay & Ali Doğan

Group: 1

Student IDs: 22067606 22067605

Date: 12.05.2023

Introduction:

The detection and localization of individuals under rubble is a critical task in search and rescue operations during natural disasters or building collapses. In such scenarios, time is of the essence, and locating survivors quickly can significantly increase their chances of survival. Traditional search and rescue methods often rely on visual cues, which can be challenging in situations where visibility is limited or obstructed.

Our Motivation:

The motivation behind this project is to develop a system that can detect and localize individuals under rubble using sound signals. Sound can travel through small openings and debris, providing an alternative means of locating survivors. By analyzing sound patterns and characteristics, we aim to create a reliable and efficient method for identifying and localizing human presence amidst the noise and chaos of a disaster site.

Literature Survey:

We conducted a thorough literature survey to explore existing research and technologies related to the detection and localization of individuals under rubble. Several studies have investigated the use of different sensors and signal processing techniques for this purpose. For example, Bozkurt, Lobaton, and Sichitiu (2016) proposed a biobotic distributed sensor network for under-rubble search and rescue, utilizing a combination of acoustic and seismic sensors. Their work demonstrated the feasibility of using sound signals for detecting trapped individuals.

Other studies have focused on developing algorithms and machine learning models to analyze sound patterns and classify different audio sources. These approaches leverage techniques such as Fast Fourier Transform (FFT), Mel-Frequency Cepstral Coefficients (MFCC), and deep neural networks to extract features and classify sound signals. By training models on labeled audio data, researchers have achieved promising results in identifying human voices, emergency vehicle sounds, and environmental noises.

Market Analysis:

The market for technologies related to search and rescue operations is growing, driven by the increasing frequency of natural disasters and the need for efficient rescue strategies. Companies and organizations involved in disaster management and emergency response are constantly seeking innovative solutions to enhance their capabilities. The detection and localization of individuals under rubble is a crucial aspect of these operations, as it directly impacts the success and effectiveness of rescue missions.

Currently, the market offers various technologies and equipment for search and rescue, including specialized cameras, drones, thermal imaging devices, and robotic systems. However, there is a lack of comprehensive solutions that specifically focus on utilizing sound signals for detecting and localizing trapped individuals. This presents an opportunity for our system to fill this gap and provide a unique and valuable solution to the market.

By developing a reliable and efficient system for detecting and localizing individuals under rubble using sound signals, we aim to contribute to the search and rescue industry and provide a tool that can save lives during critical situations. Our project's innovative approach and potential impact make it a compelling solution in the market.

transformation_between_time_and_frequency:

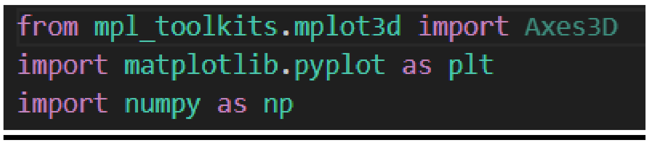
1.Importing Libraries:

- `mpl_toolkits.mplot3d` and `matplotlib.pyplot` are used for 3D plotting.

- `numpy` (imported as `np`) provides numerical computing capabilities.

- `scipy.fftpack.fft` is used for performing the Fast Fourier Transform (FFT).

- `scipy.signal.welch` is used for computing the Power Spectral Density (PSD).



```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

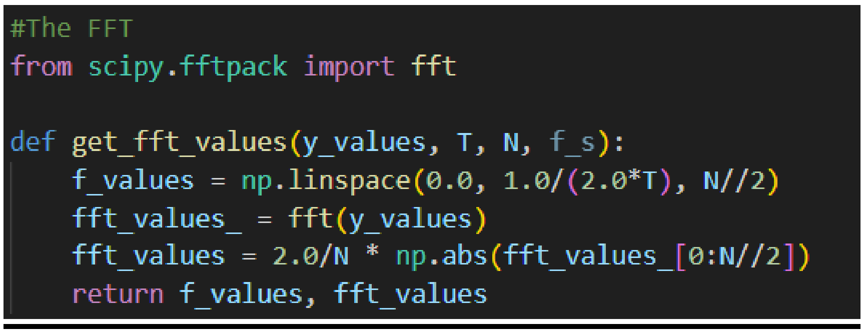
2.Defining the FFT Function:

- The function `get_fft_values` takes `y_values` (input signal), `T` (time period), `N` (number of samples), and `f_s` (sampling frequency) as parameters.

- It calculates the frequency values (`f_values`) and performs the FFT on the input signal.

- The FFT result is stored in `fft_values`.

- The function returns `f_values` and `fft_values`.



```python
#The FFT
from scipy.fftpack import fft

def get_fft_values(y_values, T, N, f_s):
    f_values = np.linspace(0.0, 1.0/(2.0*T), N//2)
    fft_values_ = fft(y_values)
    fft_values = 2.0/N * np.abs(fft_values_[0:N//2])
    return f_values, fft_values
```

3.Setting Variables and Generating Signals:

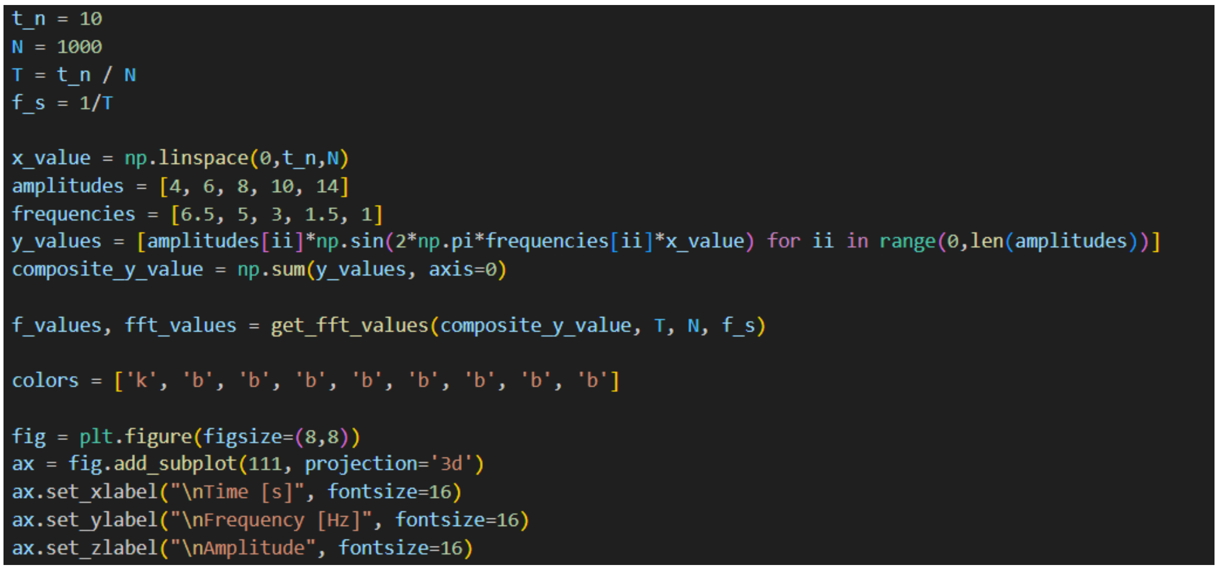
- `t_n`, `N`, `T`, and `f_s` are defined for signal parameters.

- `x_value` is generated as a time vector using `lenspiece`.

- `amplitudes` and `frequencies` represent the amplitudes and frequencies of individual sine waves.

- `y_values` is generated as a list of sine wave signals based on `amplitudes`, `frequencies`, and `x_value`.

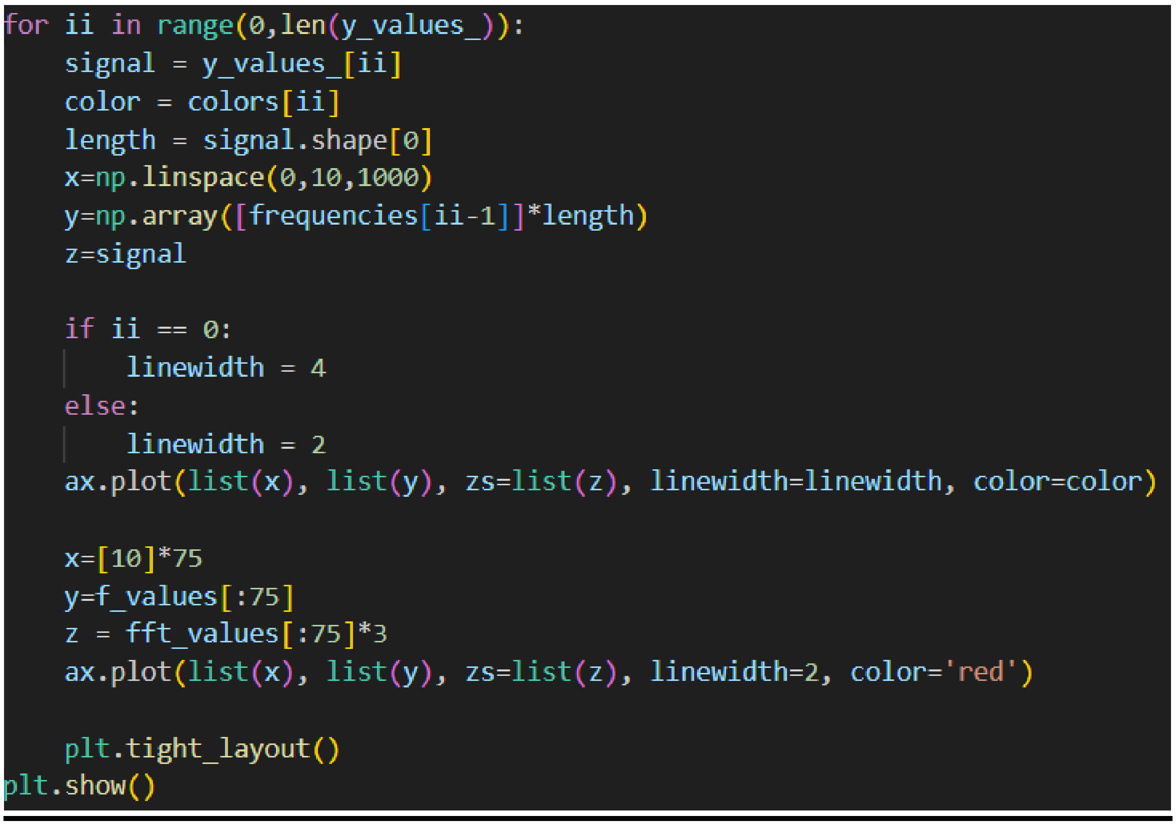
- `composite_y_value` is the sum of all individual sine waves.



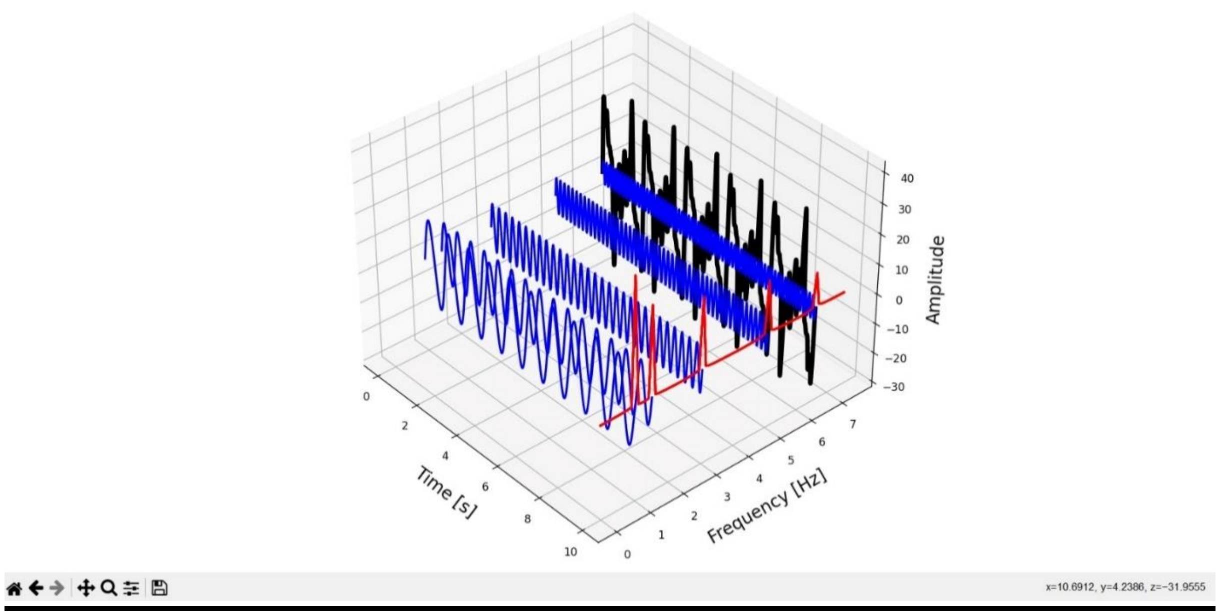
```
t_n = 10
N = 1000
T = t_n / N
f_s = 1/T

x_value = np.linspace(0,t_n,N)
amplitudes = [4, 6, 8, 10, 14]
frequencies = [6.5, 5, 3, 1.5, 1]
y_values = [amplitudes[ii]*np.sin(2*np.pi*frequencies[ii]*x_value) for ii in range(0,len(amplitudes))]
composite_y_value = np.sum(y_values, axis=0)

f_values, fft_values = get_fft_values(composite_y_value, T, N, f_s)

colors = ['k', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel("\nTime [s]", fontsize=16)
ax.set_ylabel("\nFrequency [Hz]", fontsize=16)
ax.set_zlabel("\nAmplitude", fontsize=16)
```

4.Performing FFT and Plotting 3D Signals:

- `f_values` and `fft_values` are calculated using `get_fft_values` with `composite_y_value`.
- `colors` represents the colors for different signals.
- A 3D plot is created using `fig` and `ax`.
- The labels for the axes are set.
- The loop iterates over the `y_values_` list, plots each signal, and sets the line width and color accordingly.
- The frequency domain plot is also shown as a red line.



```python
for ii in range(0,len(y_values_)):
    signal = y_values_[ii]
    color = colors[ii]
    length = signal.shape[0]
    x=np.linspace(0,10,1000)
    y=np.array([frequencies[ii-1]]*length)
    z=signal

    if ii == 0:
        linewidth = 4
    else:
        linewidth = 2
    ax.plot(list(x), list(y), zs=list(z), linewidth=linewidth, color=color)

    x=[10]*75
    y=f_values[:75]
    z = fft_values[:75]*3
    ax.plot(list(x), list(y), zs=list(z), linewidth=2, color='red')

    plt.tight_layout()
plt.show()
```

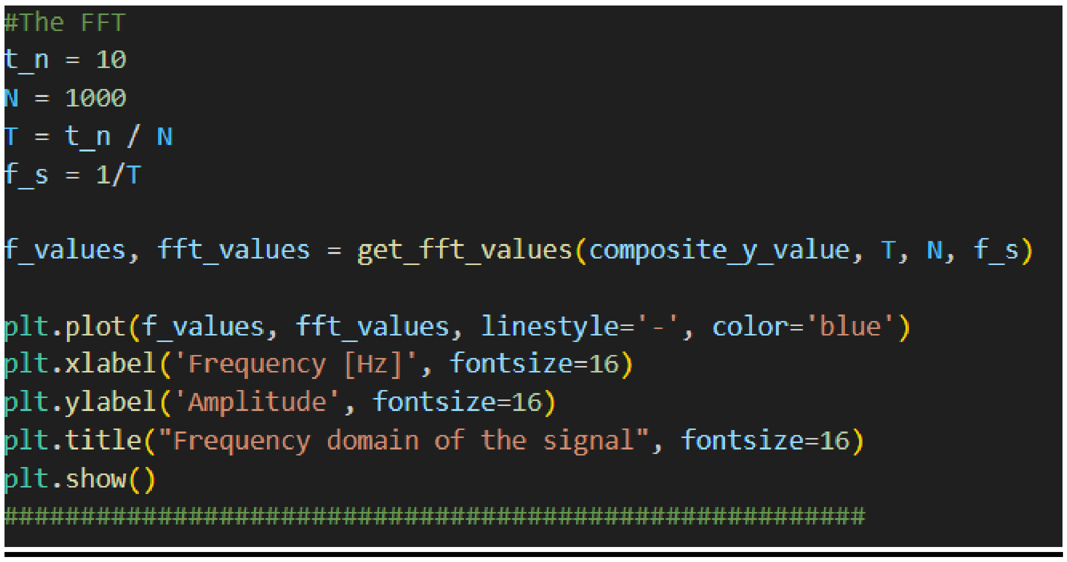


5.Plotting Frequency Domain with FFT:

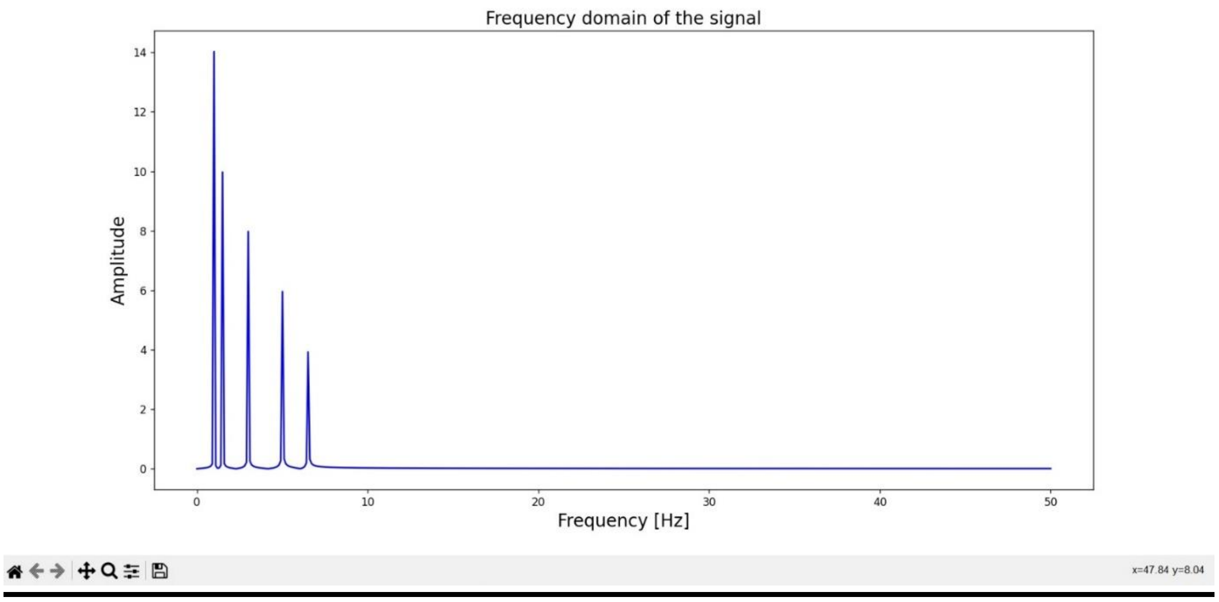
- The same FFT calculation is performed again using `get_fft_values` with `composite_y_value`.

- The frequency domain plot is created using `plt.plot` with `f_values` and `fft_values`.

- Labels and the title for the plot are set, and the plot is shown.



```
#The FFT
t_n = 10
N = 1000
T = t_n / N
f_s = 1/T

f_values, fft_values = get_fft_values(composite_y_value, T, N, f_s)

plt.plot(f_values, fft_values, linestyle='-', color='blue')
plt.xlabel('Frequency [Hz]', fontsize=16)
plt.ylabel('Amplitude', fontsize=16)
plt.title("Frequency domain of the signal", fontsize=16)
plt.show()
############################################################
```



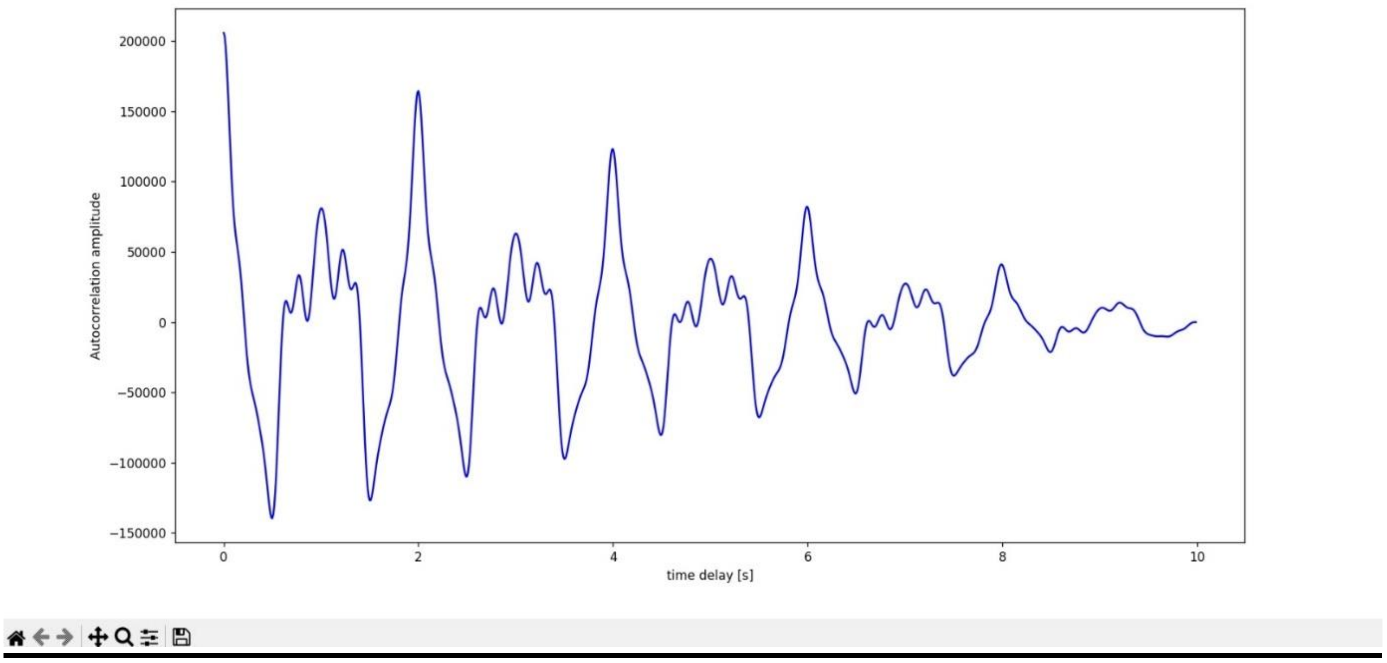
6.Performing PSD and Plotting:

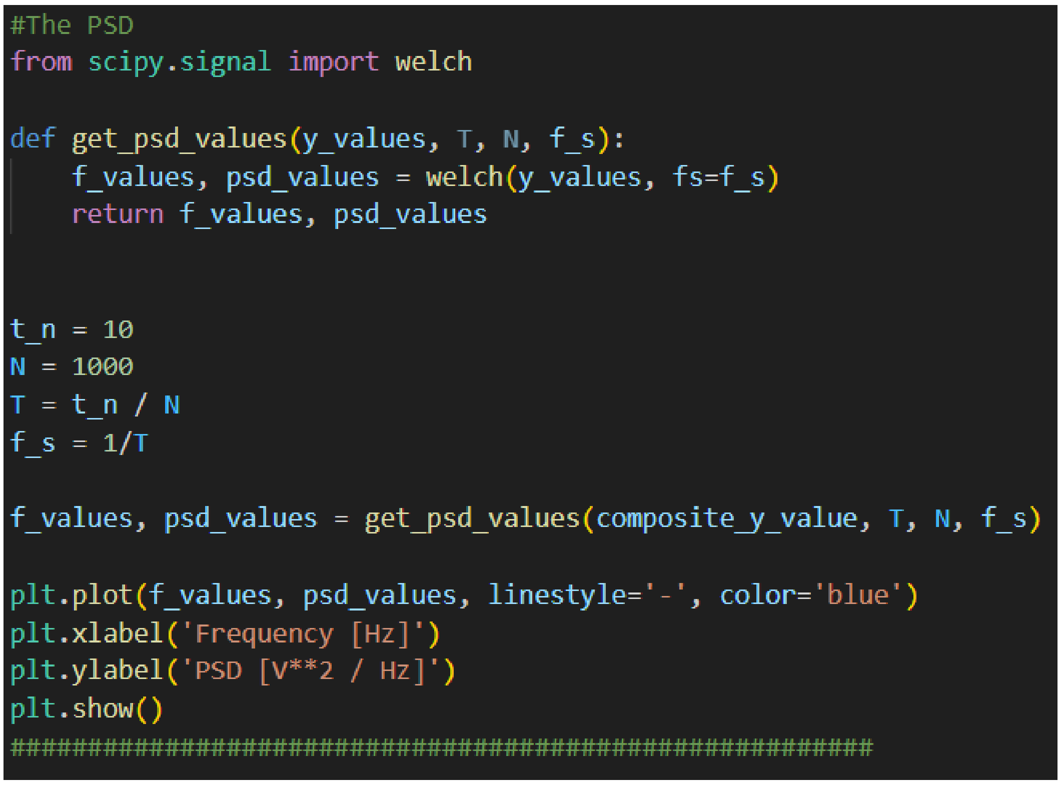
- The function `get_psd_values` takes `y_values`, `T`, `N`, and `f_s` as parameters and calculates the PSD using the Welch method.

- `f_values` and `psd_values` are obtained.

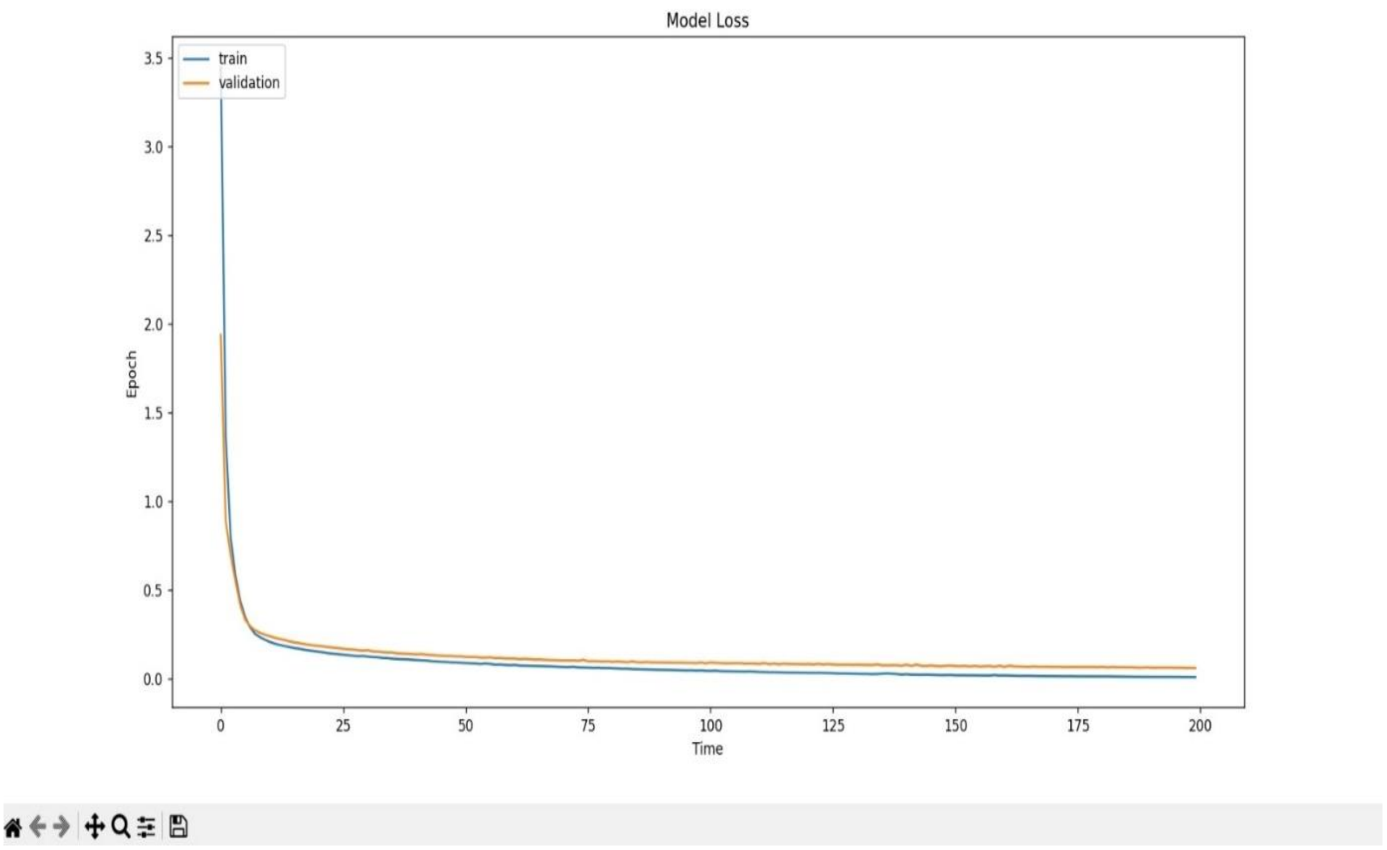
- The PSD plot is created using `plt.plot` with `f_values` and `psd_values`.

- Labels for the axes are set, and the plot is shown.



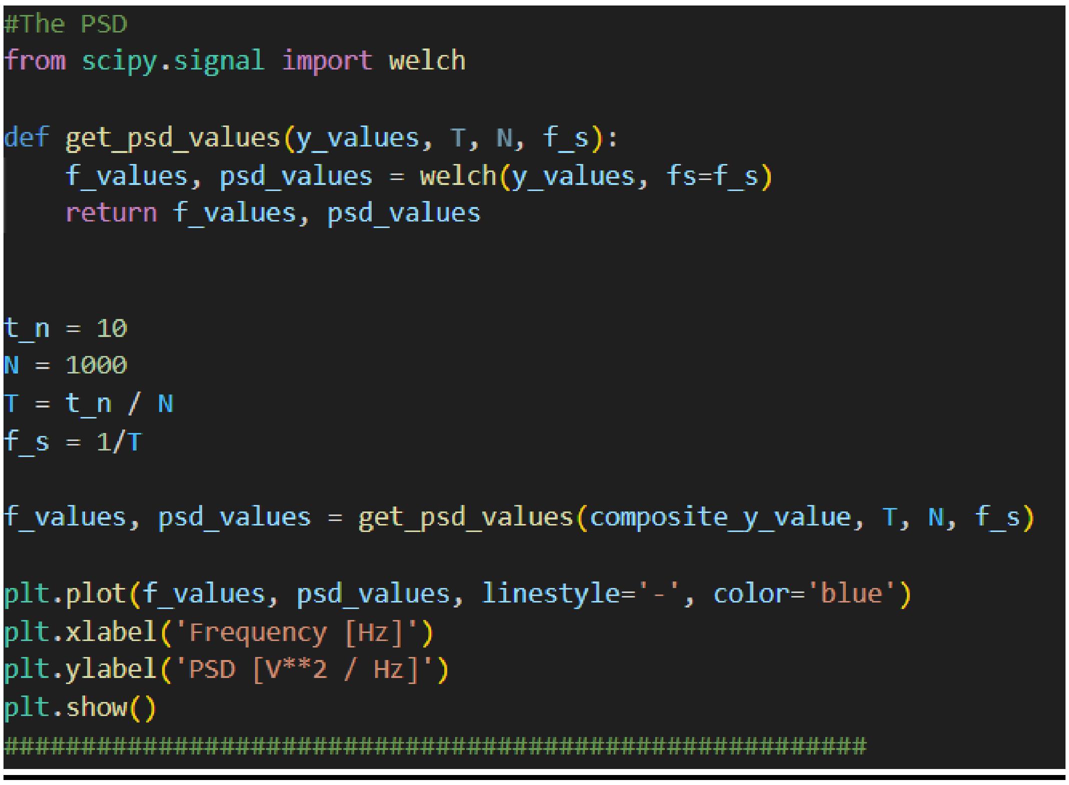


```
#The PSD
from scipy.signal import welch

def get_psd_values(y_values, T, N, f_s):
    f_values, psd_values = welch(y_values, fs=f_s)
    return f_values, psd_values


t_n = 10
N = 1000
T = t_n / N
f_s = 1/T

f_values, psd_values = get_psd_values(composite_y_value, T, N, f_s)

plt.plot(f_values, psd_values, linestyle='-', color='blue')
plt.xlabel('Frequency [Hz]')
plt.ylabel('PSD [V**2 / Hz]')
plt.show()
########################################################
```



7.Performing Autocorrelation and Plotting:

- The function `autocorr` performs auto-correlation on the input signal `x`.
- `get_autocorr_values` takes `y_values`, `T`, `N`, and `f_s` as parameters and calculates the auto-correlation values using `autocorr`.
- `t_values` and `autocorr_values` are obtained.
- The auto-correlation plot is created using `plt.plot` with `t_values` and `autocorr_values`.
- Labels for the axes are set, and the plot is shown.

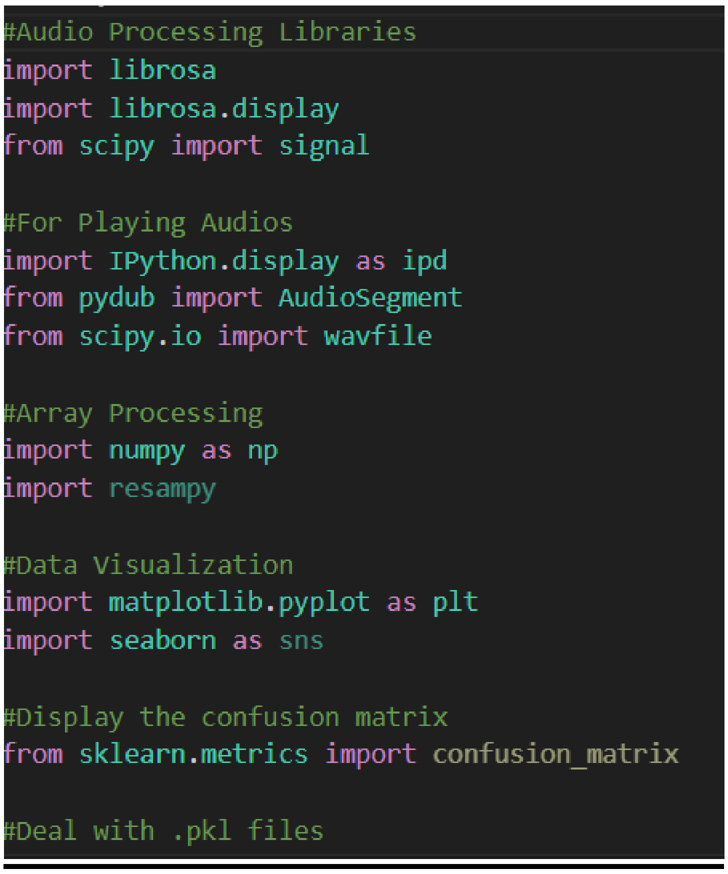


```python
#The PSD
from scipy.signal import welch

def get_psd_values(y_values, T, N, f_s):
    f_values, psd_values = welch(y_values, fs=f_s)
    return f_values, psd_values


t_n = 10
N = 1000
T = t_n / N
f_s = 1/T

f_values, psd_values = get_psd_values(composite_y_value, T, N, f_s)

plt.plot(f_values, psd_values, linestyle='-', color='blue')
plt.xlabel('Frequency [Hz]')
plt.ylabel('PSD [V**2 / Hz]')
plt.show()
########################################################
```

The overall purpose of the code is to analyze a composite signal by performing the Fast Fourier Transform (FFT), computing the Power Spectral Density (PSD), and calculating the autocorrelation. The results are visualized using various plots, including 3D signal representation, frequency domain plot, PSD plot, and auto-correlation plot. These analyses provide insights into the frequency content and characteristics of the signal.

analysing_human_sound:

1.Importing Libraries: The code begins by importing various libraries required for audio processing, data visualization, machine learning, and file manipulation. These libraries include librosa, scipy, IPython.display, pydub, numpy, resampy, matplotlib.pyplot, seaborn, sklearn.metrics, pickle, pandas, keras.utils, and sklearn.model_selection.



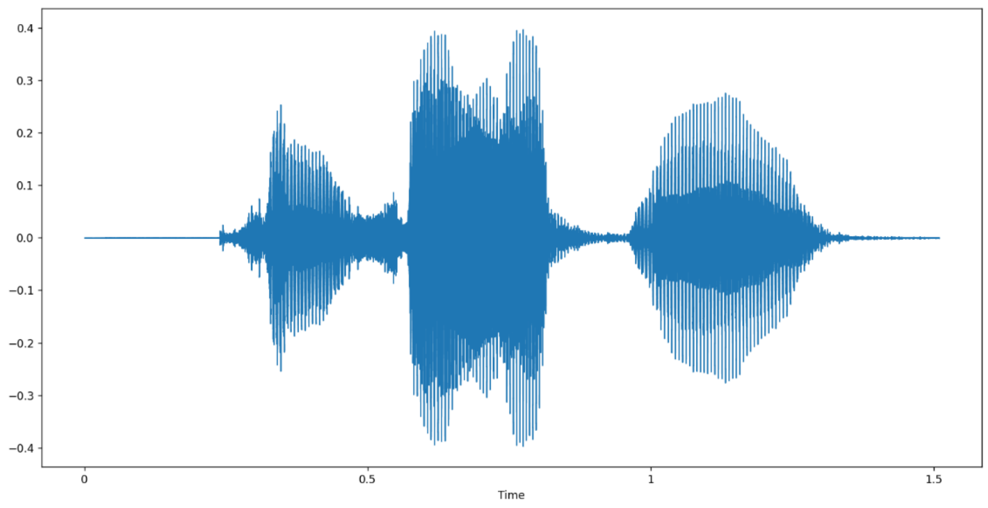
```
#Audio Processing Libraries
import librosa
import librosa.display
from scipy import signal

#For Playing Audios
import IPython.display as ipd
from pydub import AudioSegment
from scipy.io import wavfile

#Array Processing
import numpy as np
import resampy

#Data Visualization
import matplotlib.pyplot as plt
import seaborn as sns

#Display the confusion matrix
from sklearn.metrics import confusion_matrix

#Deal with .pkl files
```

2.Loading and Visualizing Audio: The code loads an audio file using the `librosa.load` function and visualizes the waveform using `librosa.display.waveshow` and `matplotlib.pyplot`. It also plays the audio using `IPython.display.Audio`.

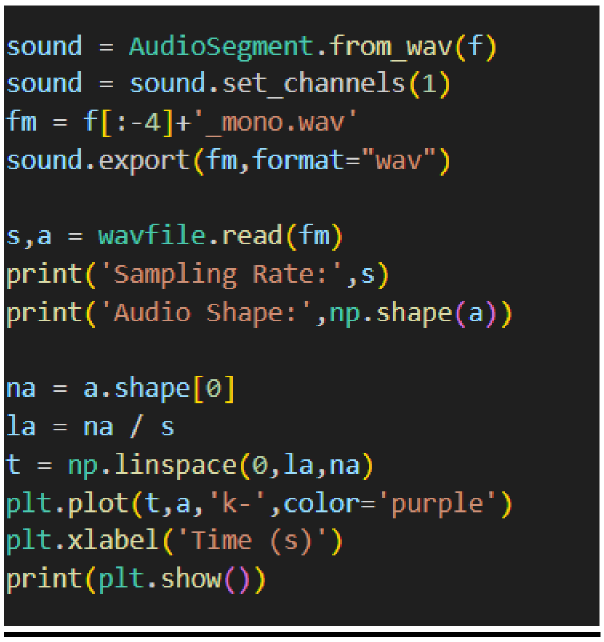
```
filename = "input/emergency_vehicle_sounds/sounds/human/sound_601.wav"
plt.figure(figsize=(14,5))
data, sample_rate = librosa.load(filename)
librosa.display.waveshow(data, sr=sample_rate)
ipd.Audio(filename)
print(plt.show())
```

3.Reading Audio and Information: The code reads the audio file using `scipy.io.wavfile.read` and displays the sampling rate (`s`) and the shape of the audio signal (`a`). It also calculates the audio time duration (`la`) based on the number of samples and sampling rate.

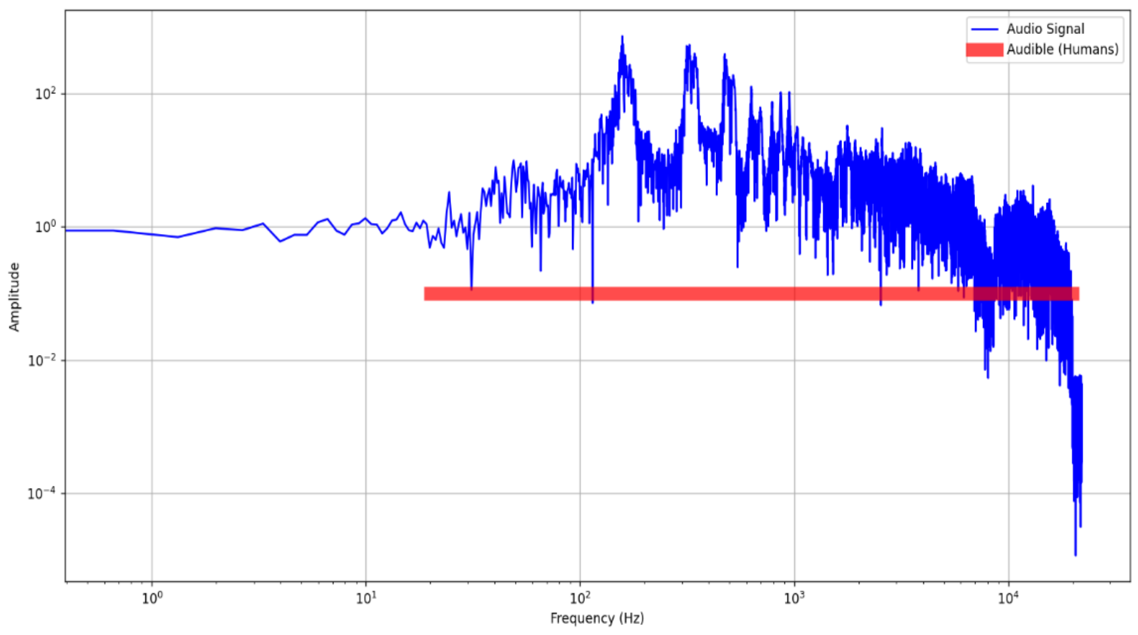
```
#Import the .wav audio
f = "input/emergency_vehicle_sounds/sounds/human/sound_601.wav"
#s = sampling (int)
#a = audio signal (numpy array)
s,a = wavfile.read(f)
print('Sampling Rate:',s)
print('Audio Shape:',np.shape(a))

#number of samples
na = a.shape[0]
#audio time duration
la = na / s
```



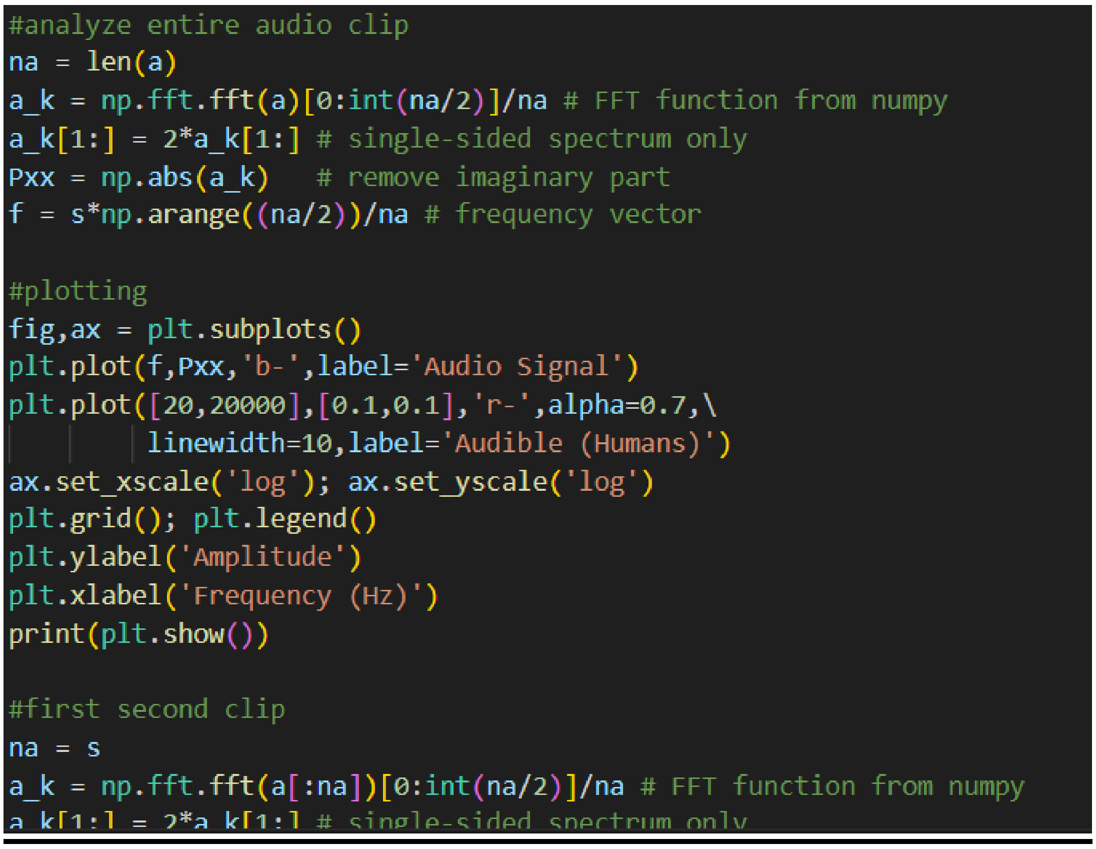
4.Converting Audio to Mono: The code converts the stereo audio signal to mono using the `pydub.AudioSegment` library and exports it as a mono WAV file. The new file is then read again to update the sampling rate (`s`) and audio signal (`a`).



```
sound = AudioSegment.from_wav(f)
sound = sound.set_channels(1)
fm = f[:-4]+'_mono.wav'
sound.export(fm,format="wav")

s,a = wavfile.read(fm)
print('Sampling Rate:',s)
print('Audio Shape:',np.shape(a))

na = a.shape[0]
la = na / s
t = np.linspace(0,la,na)
plt.plot(t,a,'k-',color='purple')
plt.xlabel('Time (s)')
print(plt.show())
```

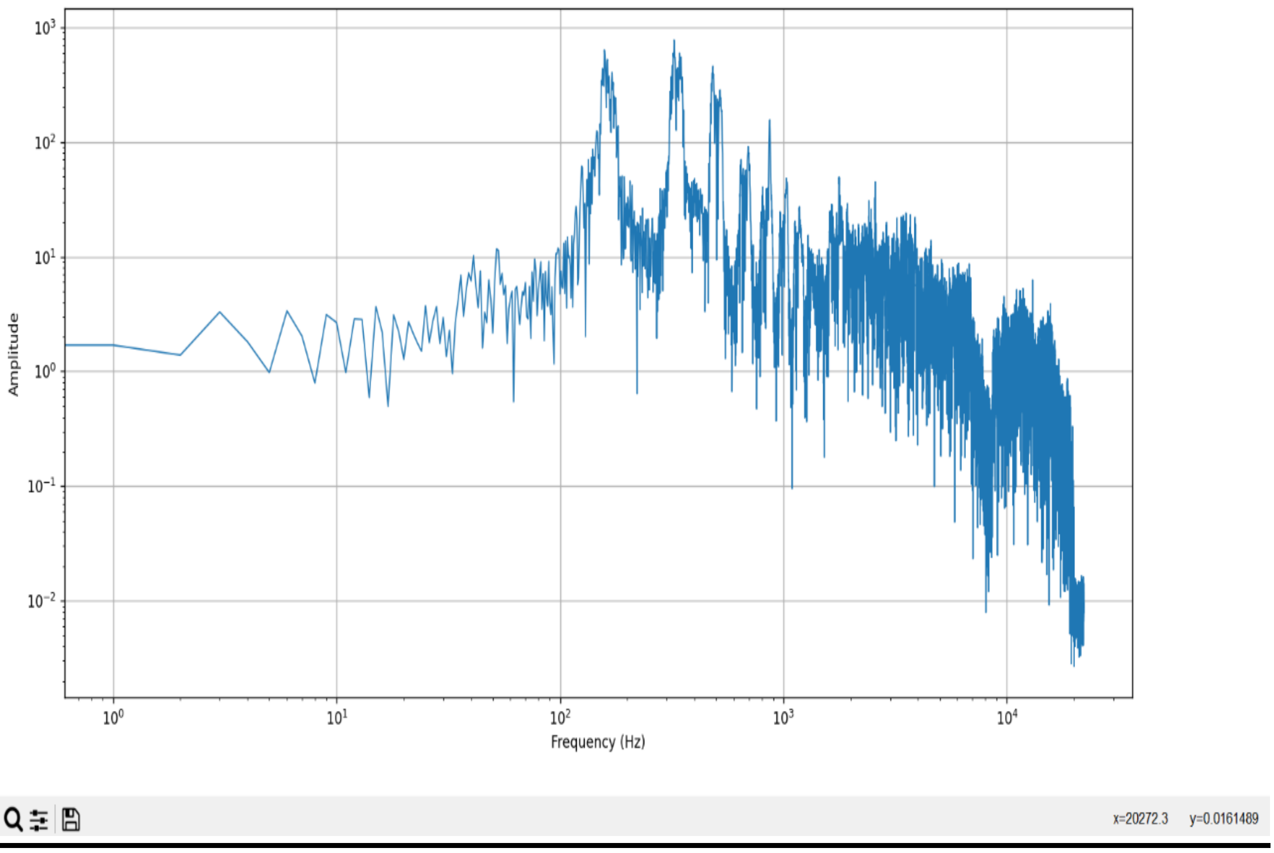


5.Visualizing Mono Audio: The code plots the mono audio signal over time using `matplotlib.pyplot.plot` and `numpy.linspace`. The x-axis represents time in seconds, and the y-axis represents the amplitude of the audio signal.

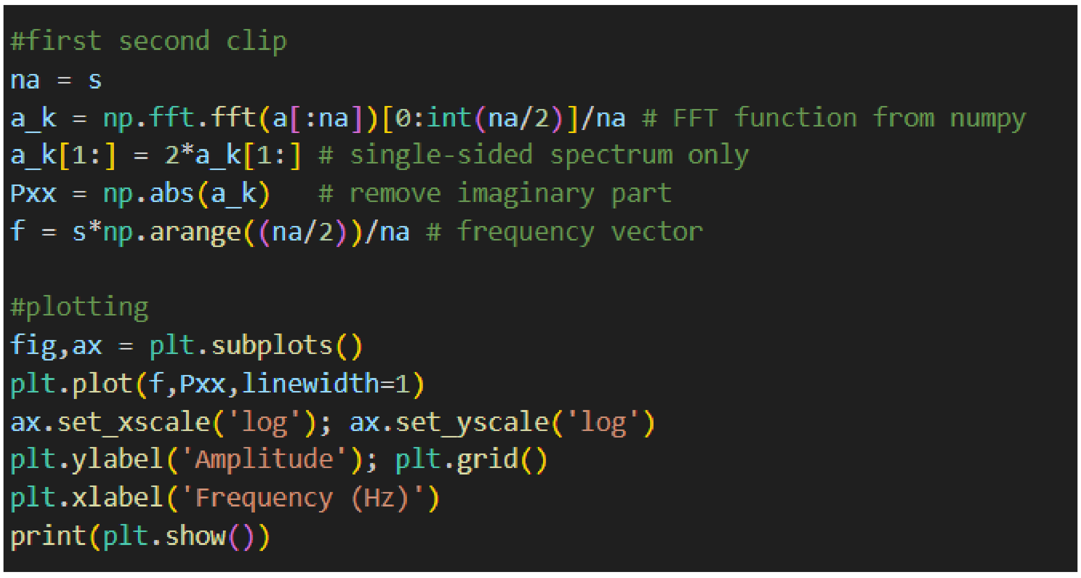


```
#analyze entire audio clip
na = len(a)
a_k = np.fft.fft(a)[0:int(na/2)]/na # FFT function from numpy
a_k[1:] = 2*a_k[1:] # single-sided spectrum only
Pxx = np.abs(a_k)   # remove imaginary part
f = s*np.arange((na/2))/na # frequency vector

#plotting
fig,ax = plt.subplots()
plt.plot(f,Pxx,'b-',label='Audio Signal')
plt.plot([20,20000],[0.1,0.1],'r-',alpha=0.7,\
         linewidth=10,label='Audible (Humans)')
ax.set_xscale('log'); ax.set_yscale('log')
plt.grid(); plt.legend()
plt.ylabel('Amplitude')
plt.xlabel('Frequency (Hz)')
print(plt.show())

#first second clip
na = s
a_k = np.fft.fft(a[:na])[0:int(na/2)]/na # FFT function from numpy
a_k[1:] = 2*a_k[1:] # single-sided spectrum only
```

6.Audio Spectrum Analysis: The code performs frequency analysis on the entire audio clip using the Fast Fourier Transform (FFT) provided by `numpy.fft.fft`. It computes the magnitude spectrum (`Pxx`) and frequency vector (`f`) based on the FFT output. The single-sided spectrum is obtained by doubling the amplitudes of all frequency bins except the DC component. The resulting spectrum is plotted using `matplotlib.pyplot.plot`.

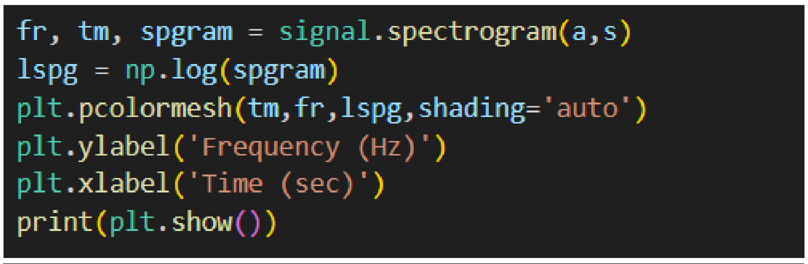


7.First Second Clip Analysis: The code performs frequency analysis on the first second of the audio clip. It follows a similar procedure as in the previous step but only considers the first `s` samples of the audio signal. The resulting spectrum is plotted using `matplotlib.pyplot.plot`.

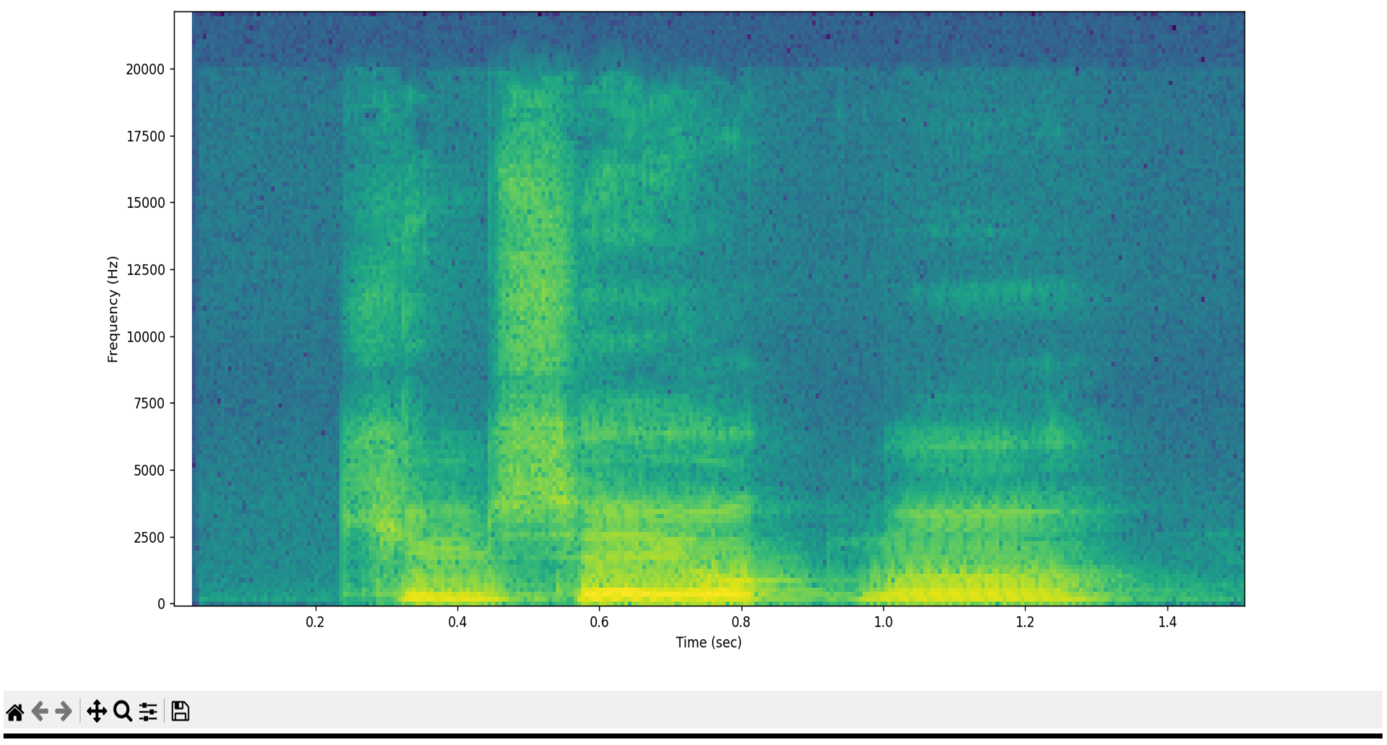


```
#first second clip
na = s
a_k = np.fft.fft(a[:na])[0:int(na/2)]/na # FFT function from numpy
a_k[1:] = 2*a_k[1:] # single-sided spectrum only
Pxx = np.abs(a_k)   # remove imaginary part
f = s*np.arange((na/2))/na # frequency vector

#plotting
fig,ax = plt.subplots()
plt.plot(f,Pxx,linewidth=1)
ax.set_xscale('log'); ax.set_yscale('log')
plt.ylabel('Amplitude'); plt.grid()
plt.xlabel('Frequency (Hz)')
print(plt.show())
```

8.Spectrogram: The code generates a spectrogram of the audio signal using the `signal.spectrogram` function from `scipy`. It calculates the spectrogram (`spgram`) and takes its logarithm to obtain the log-scaled representation (`lspg`). The spectrogram is visualized using `matplotlib.pyplot.pcolormesh`.

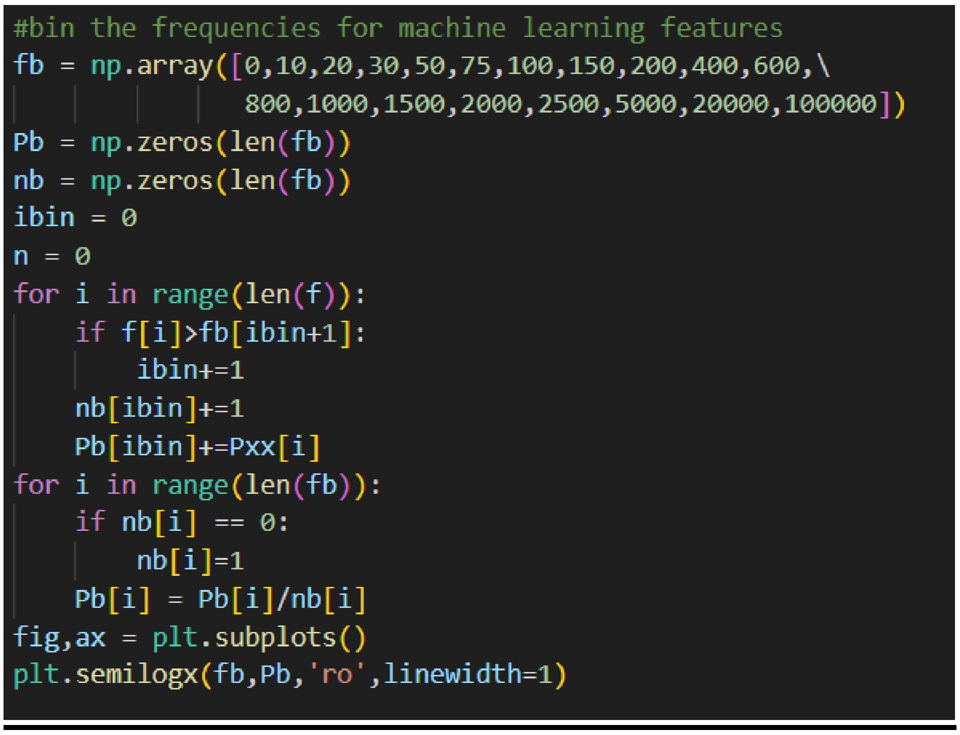


```
fr, tm, spgram = signal.spectrogram(a,s)
lspg = np.log(spgram)
plt.pcolormesh(tm,fr,lspg,shading='auto')
plt.ylabel('Frequency (Hz)')
plt.xlabel('Time (sec)')
print(plt.show())
```

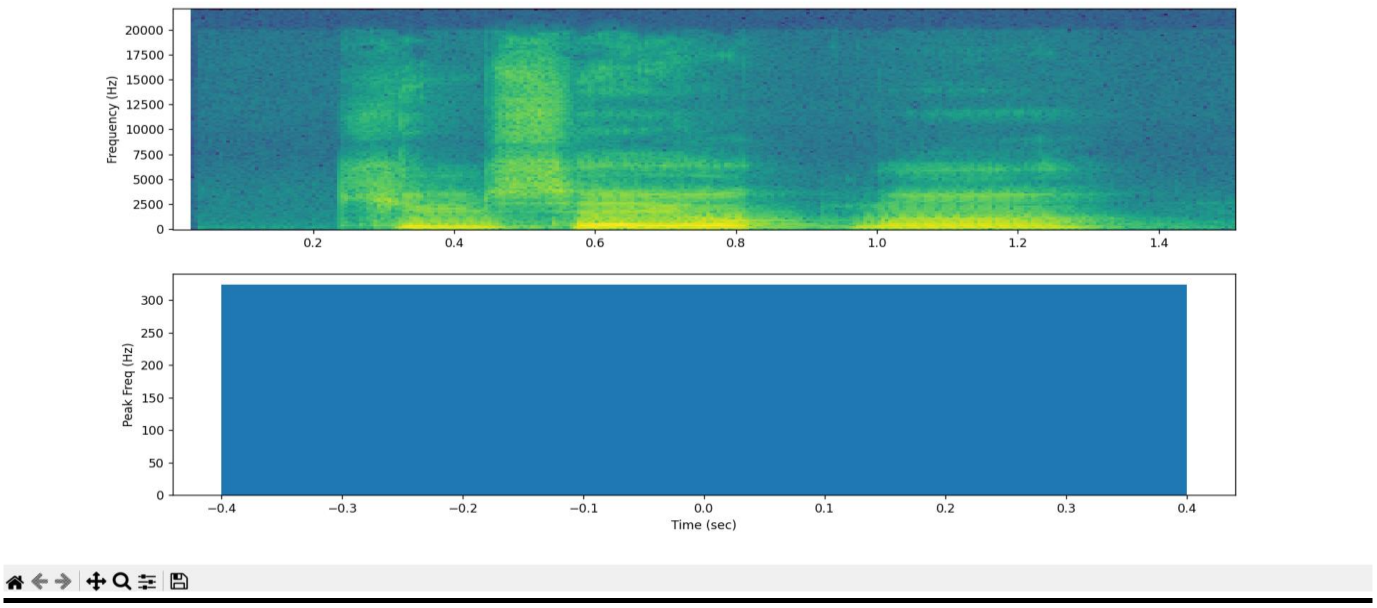


Frequency (Hz)
20000
17500
15000
12500
10000
7500
5000
2500
0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Time (sec)

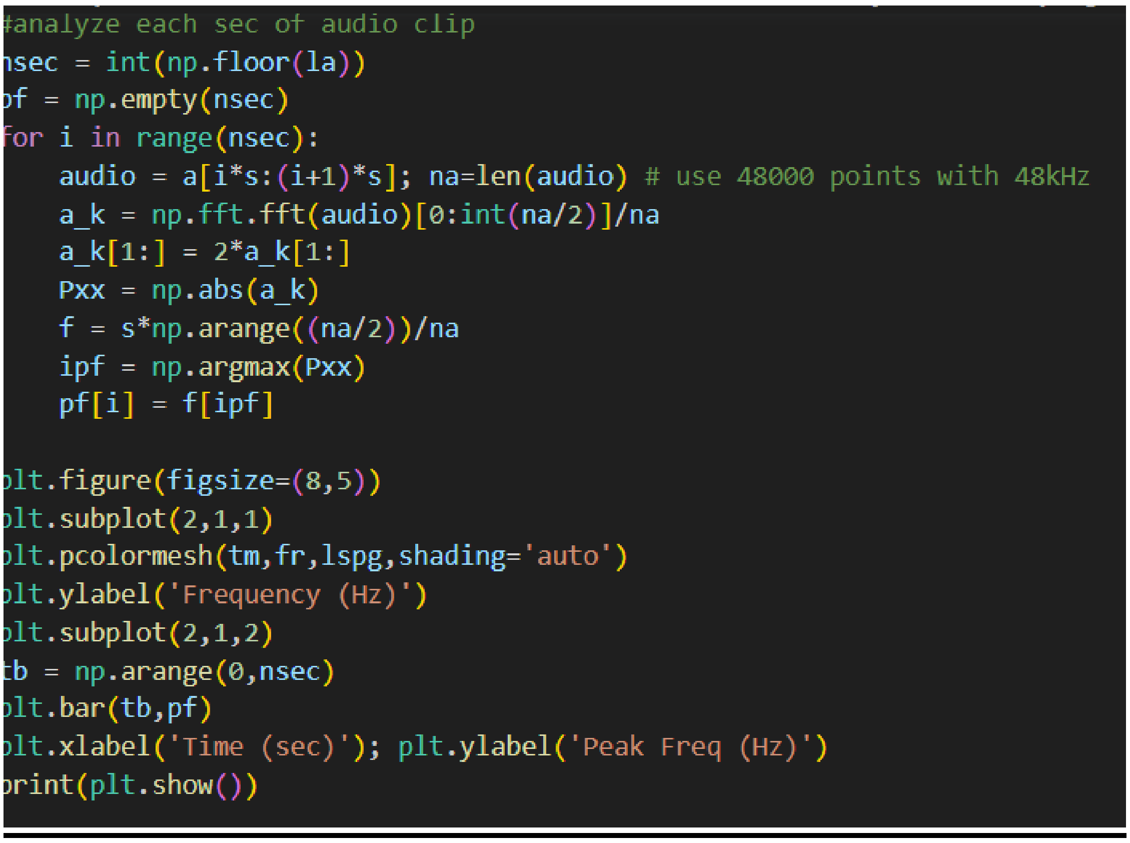
9.Binning Frequencies: The code defines frequency bins (`fb`) for machine learning features. It initializes arrays (`Pb` and `nb`) to store average power and count of frequency occurrences for each bin. It then iterates over the frequency vector (`f`) and accumulates the power values for each bin. Finally, it normalizes the power values by the count and plots them using `matplotlib.pyplot.semilogx`.



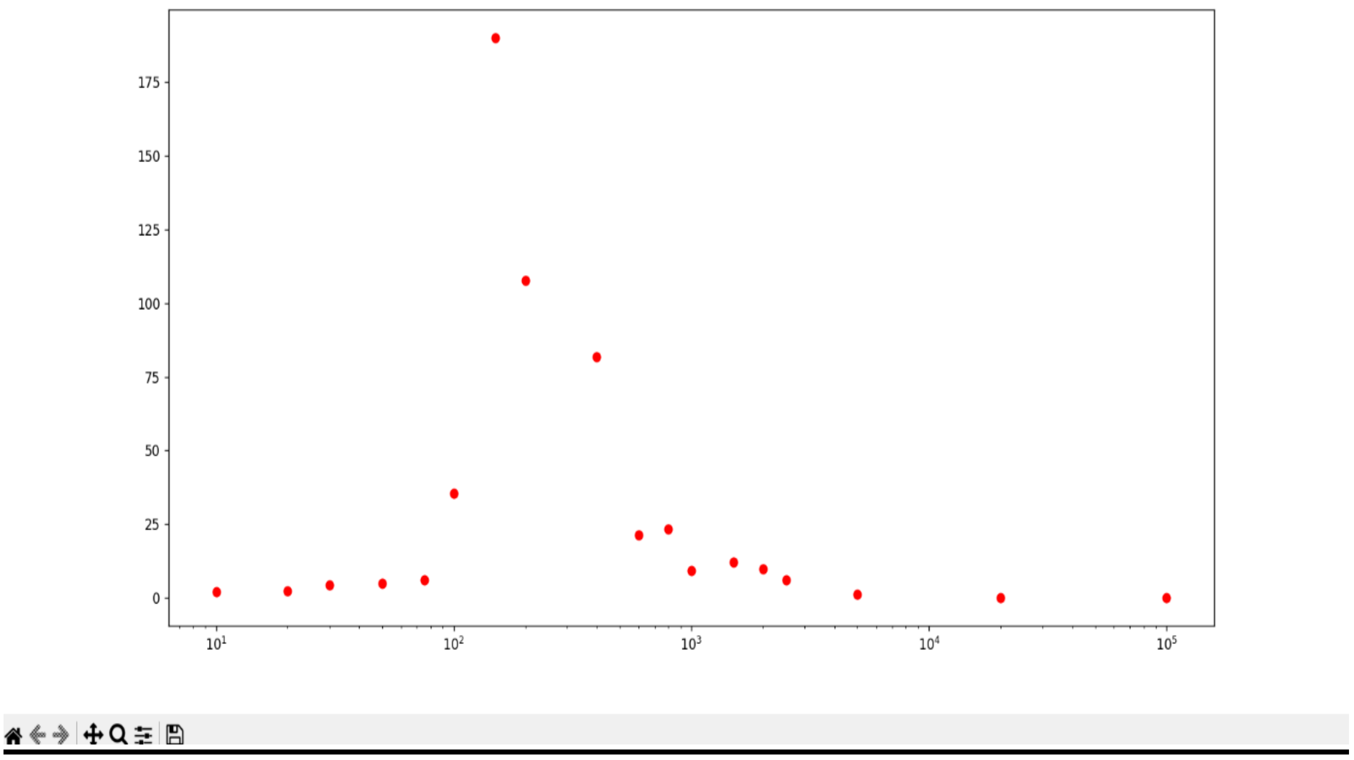
```
#bin the frequencies for machine learning features
fb = np.array([0,10,20,30,50,75,100,150,200,400,600,\
               800,1000,1500,2000,2500,5000,20000,100000])
Pb = np.zeros(len(fb))
nb = np.zeros(len(fb))
ibin = 0
n = 0
for i in range(len(f)):
    if f[i]>fb[ibin+1]:
        ibin+=1
    nb[ibin]+=1
    Pb[ibin]+=Pxx[i]
for i in range(len(fb)):
    if nb[i] == 0:
        nb[i]=1
    Pb[i] = Pb[i]/nb[i]
fig,ax = plt.subplots()
plt.semilogx(fb,Pb,'ro',linewidth=1)
```



10.Analysis of Each Second of Audio: The code performs frequency analysis on each second of the audio clip. It divides the audio signal into consecutive one-second segments and calculates the peak frequency for each segment. The peak frequencies are stored in an array (`pf`). The spectrogram and peak frequencies are visualized using `matplotlib.pyplot.pcolormesh` and `matplotlib.pyplot.bar` respectively.



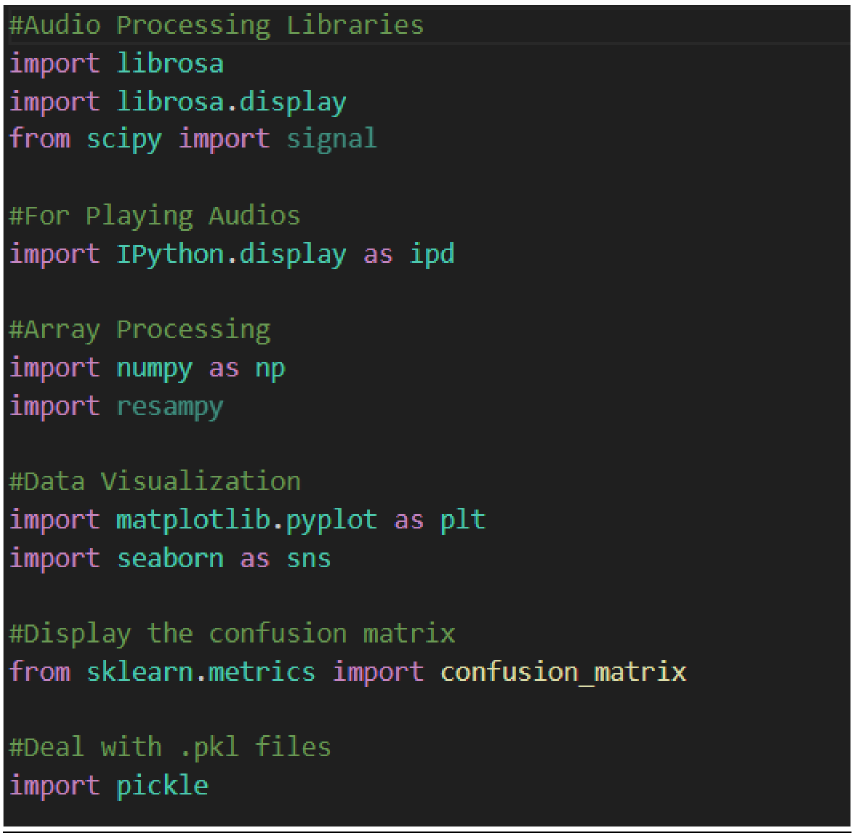
```
#analyze each sec of audio clip
nsec = int(np.floor(la))
pf = np.empty(nsec)
for i in range(nsec):
    audio = a[i*s:(i+1)*s]; na=len(audio) # use 48000 points with 48kHz
    a_k = np.fft.fft(audio)[0:int(na/2)]/na
    a_k[1:] = 2*a_k[1:]
    Pxx = np.abs(a_k)
    f = s*np.arange((na/2))/na
    ipf = np.argmax(Pxx)
    pf[i] = f[ipf]

plt.figure(figsize=(8,5))
plt.subplot(2,1,1)
plt.pcolormesh(tm,fr,lspg,shading='auto')
plt.ylabel('Frequency (Hz)')
plt.subplot(2,1,2)
tb = np.arange(0,nsec)
plt.bar(tb,pf)
plt.xlabel('Time (sec)'); plt.ylabel('Peak Freq (Hz)')
print(plt.show())
```



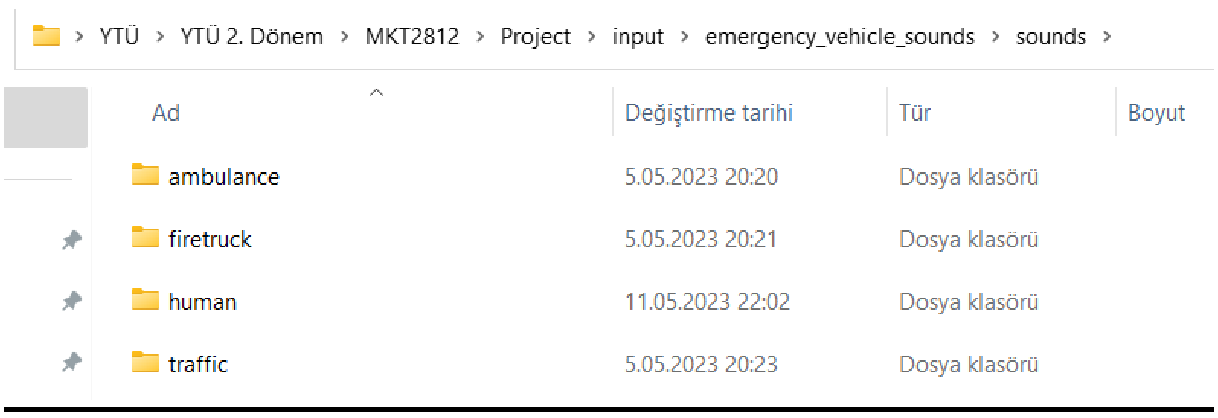
Overall, the code focuses on loading, visualizing, and analyzing audio signals using various techniques such as waveform display, spectrogram generation, frequency analysis, and feature extraction. It utilizes libraries like librosa, scipy, pydub, numpy, and matplotlib for these tasks. Additionally, it includes some data preprocessing and visualization operations to prepare the audio data for further analysis or machine learning tasks. The code provides insights into the characteristics of the audio signal, such as its amplitude over time, frequency components, and peak frequencies at different time intervals.

emergency_vehicles_sound_classification:

1.Audio Processing Libraries: We import various libraries such as librosa, scipy, IPython.display, numpy, resampy, matplotlib.pyplot, seaborn, etc., which are commonly used for audio processing tasks.

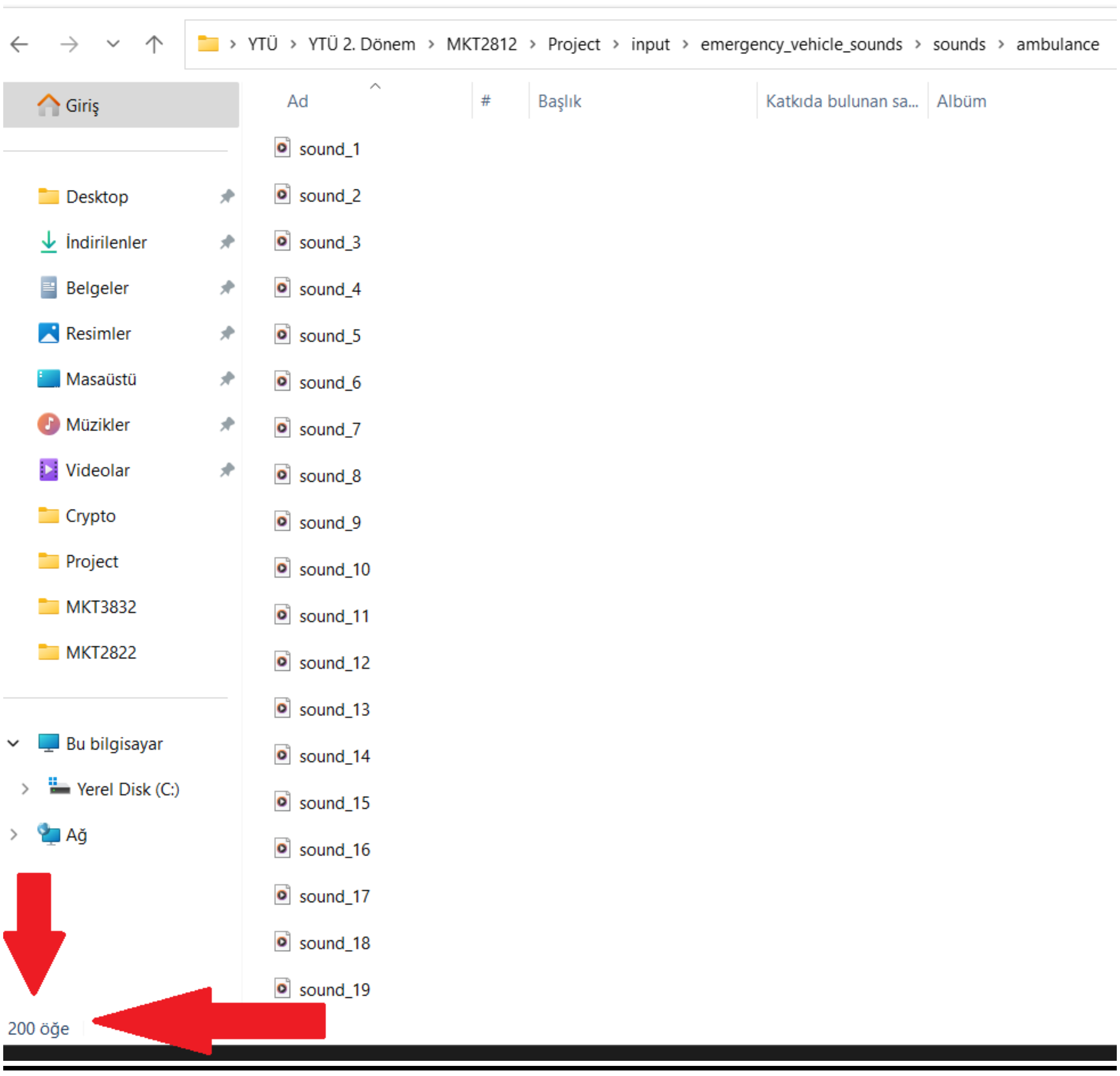


```
#Audio Processing Libraries
import librosa
import librosa.display
from scipy import signal

#For Playing Audios
import IPython.display as ipd

#Array Processing
import numpy as np
import resampy

#Data Visualization
import matplotlib.pyplot as plt
import seaborn as sns

#Display the confusion matrix
from sklearn.metrics import confusion_matrix

#Deal with .pkl files
import pickle
```



› YTÜ › YTÜ 2. Dönem › MKT2812 › Project › input › emergency_vehicle_sounds › sounds ›

Ad	Değiştirme tarihi	Tür	Boyut
ambulance	5.05.2023 20:20	Dosya klasörü	
firetruck	5.05.2023 20:21	Dosya klasörü	
human	11.05.2023 22:02	Dosya klasörü	
traffic	5.05.2023 20:23	Dosya klasörü	



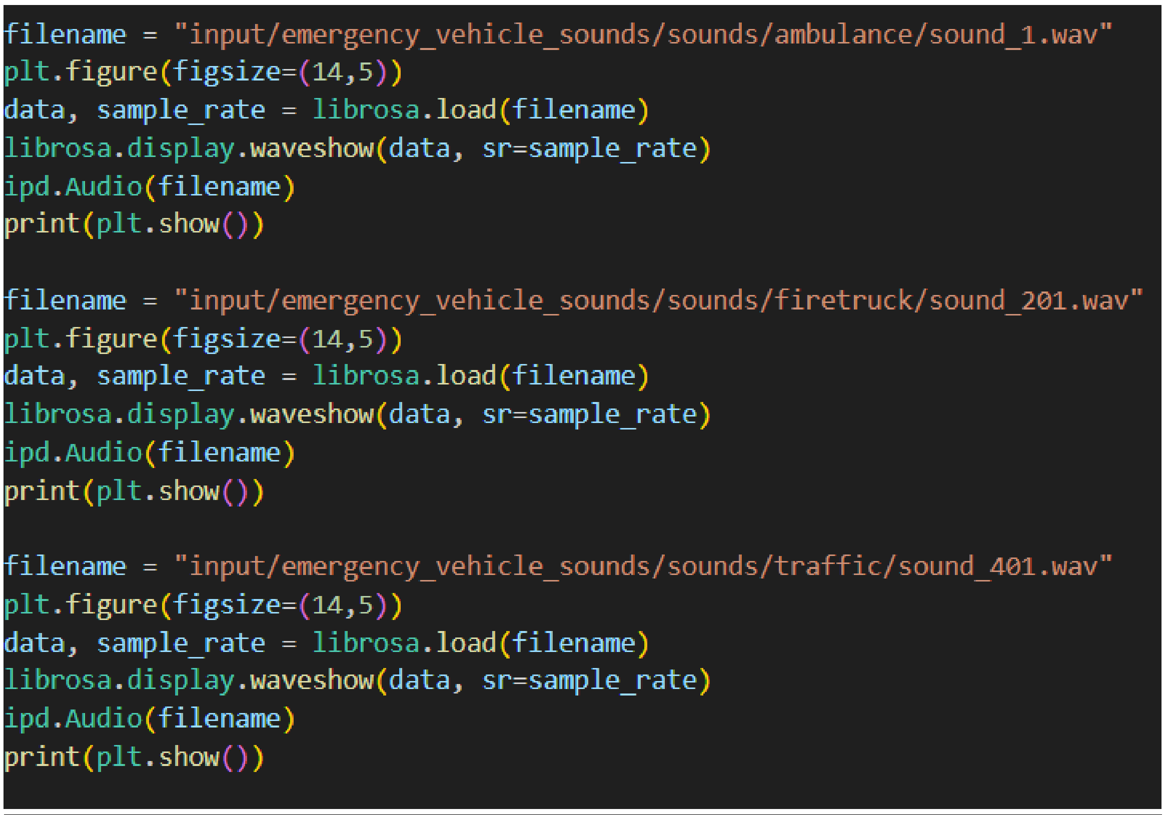
› YTÜ › YTÜ 2. Dönem › MKT2812 › Project › input › emergency_vehicle_sounds › sounds › ambulance

Ad	#	Başlık	Katkıda bulunan sa...	Albüm
sound_1				
sound_2				
sound_3				
sound_4				
sound_5				
sound_6				
sound_7				
sound_8				
sound_9				
sound_10				
sound_11				
sound_12				
sound_13				
sound_14				
sound_15				
sound_16				
sound_17				
sound_18				
sound_19				

200 öğe

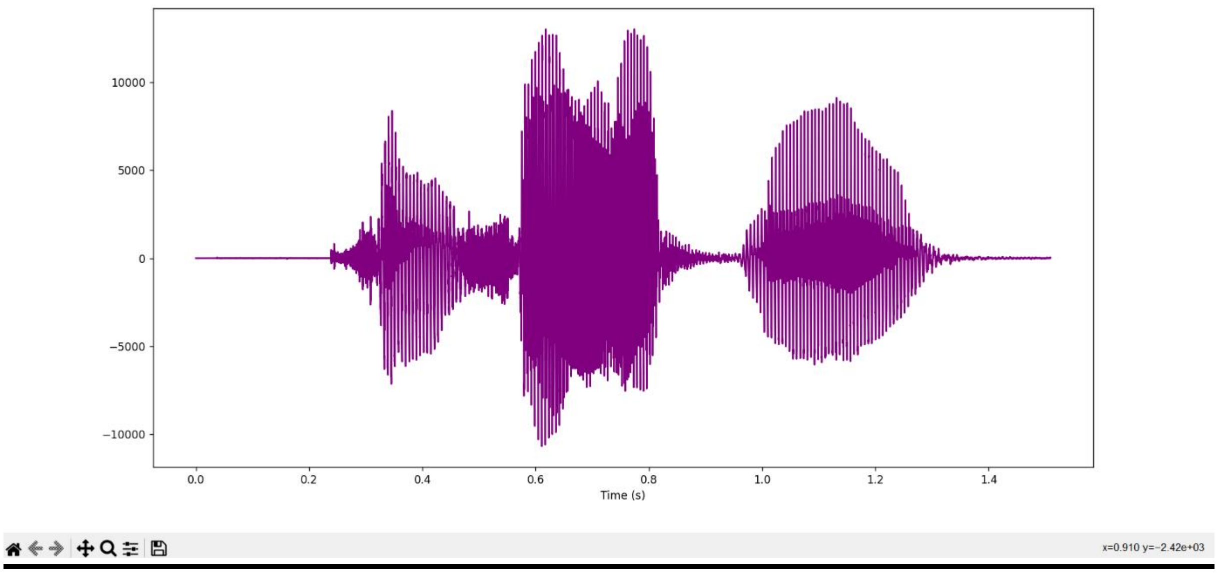
We found datasets containing open world audio files from Kaggle and used datasets consisting of 200 audio files each.

2.Data Visualization: We import libraries like matplotlib.pyplot and seaborn for visualizing audio waveforms, spectrograms, and confusion matrices.

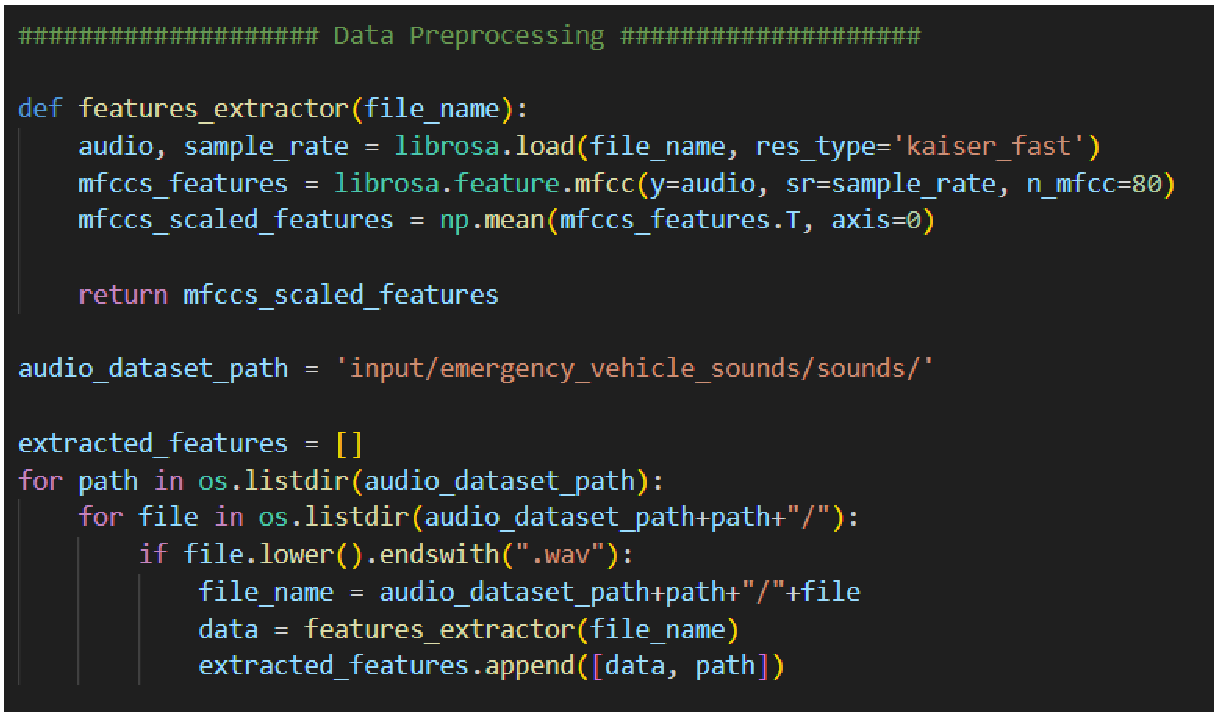


```
filename = "input/emergency_vehicle_sounds/sounds/ambulance/sound_1.wav"
plt.figure(figsize=(14,5))
data, sample_rate = librosa.load(filename)
librosa.display.waveshow(data, sr=sample_rate)
ipd.Audio(filename)
print(plt.show())

filename = "input/emergency_vehicle_sounds/sounds/firetruck/sound_201.wav"
plt.figure(figsize=(14,5))
data, sample_rate = librosa.load(filename)
librosa.display.waveshow(data, sr=sample_rate)
ipd.Audio(filename)
print(plt.show())

filename = "input/emergency_vehicle_sounds/sounds/traffic/sound_401.wav"
plt.figure(figsize=(14,5))
data, sample_rate = librosa.load(filename)
librosa.display.waveshow(data, sr=sample_rate)
ipd.Audio(filename)
print(plt.show())
```

3.Loading and Displaying Audio Files: We load audio files using librosa’s `load` function and visualize the waveform using `waveshow` and `Audio` functions from librosa and IPython.display, respectively. This allows us to inspect the audio data visually and audibly.

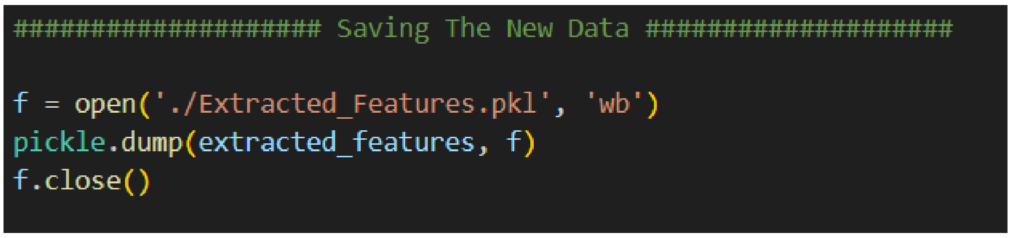


4.Data Preprocessing: We define a function `features_extractor` that takes a file name as input and extracts features from the audio using librosa. Specifically, we compute Mel-Frequency Cepstral Coefficients (MFCCs), which are commonly used for audio feature extraction. The extracted features are then scaled and returned.



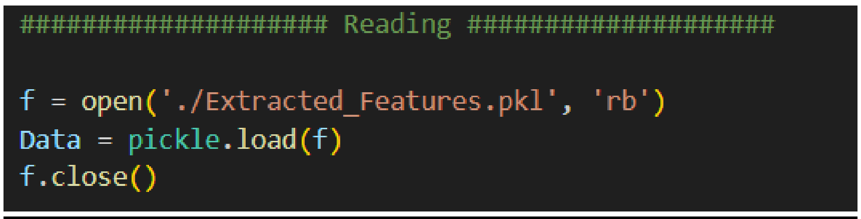
```python
#################### Data Preprocessing ####################

def features_extractor(file_name):
    audio, sample_rate = librosa.load(file_name, res_type='kaiser_fast')
    mfccs_features = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=80)
    mfccs_scaled_features = np.mean(mfccs_features.T, axis=0)

    return mfccs_scaled_features

audio_dataset_path = 'input/emergency_vehicle_sounds/sounds/'

extracted_features = []
for path in os.listdir(audio_dataset_path):
    for file in os.listdir(audio_dataset_path+path+"/"):
        if file.lower().endswith(".wav"):
            file_name = audio_dataset_path+path+"/"+file
            data = features_extractor(file_name)
            extracted_features.append([data, path])
```

5.Creating a Dataset: We create an empty list called `extracted_features` to store the extracted features and their corresponding classes. We iterate over the files in the audio dataset path, extract features using the `features_extractor` function and append them to the `extracted_features` list along with their respective classes.



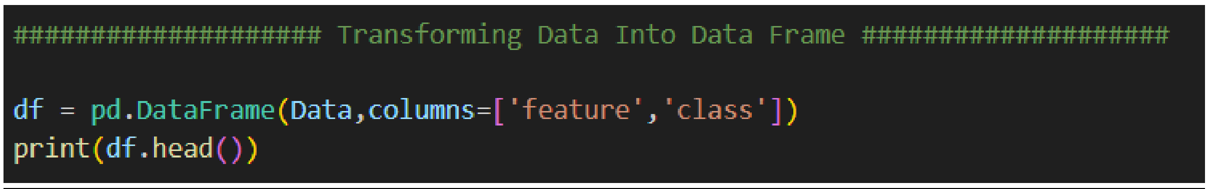
```python
#################### Saving The New Data ####################

f = open('./Extracted_Features.pkl', 'wb')
pickle.dump(extracted_features, f)
f.close()
```

6.Saving and Reading Data: We save the extracted features to a pickle file using the `pickle.dump` function. Later, we read the saved data from the pickle file using `pickle.load`.



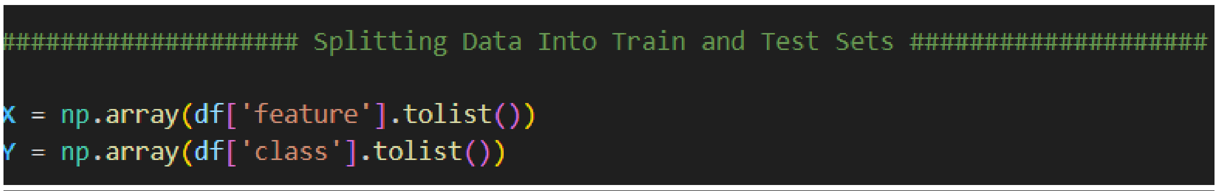
```
#################### Reading ####################

f = open('./Extracted_Features.pkl', 'rb')
Data = pickle.load(f)
f.close()
```

7.Transforming Data into DataFrame: We create a DataFrame called `df` to store the extracted features and their corresponding classes. This allows for convenient data manipulation and analysis.



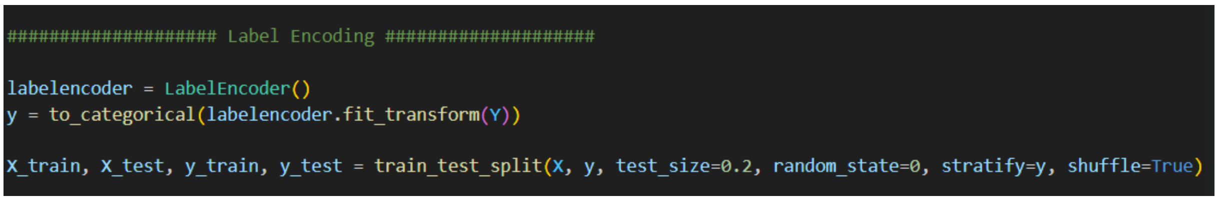
```
#################### Transforming Data Into Data Frame ####################

df = pd.DataFrame(Data,columns=['feature','class'])
print(df.head())
```

8.Splitting Data into Train and Test Sets: We split the data into training and testing sets using the `train_test_split` function from scikit-learn. The input features (`X`) and target classes (`y`) are converted to NumPy arrays.



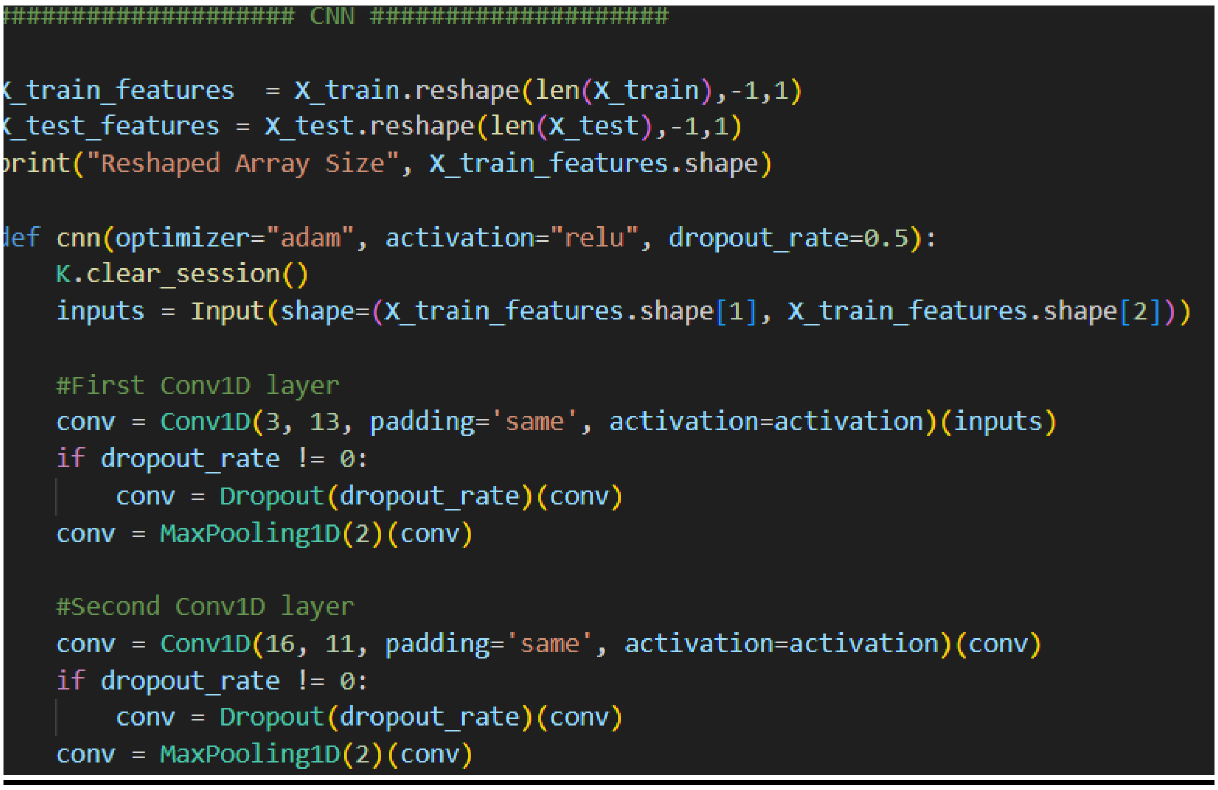
```
#################### Splitting Data Into Train and Test Sets ####################

X = np.array(df['feature'].tolist())
Y = np.array(df['class'].tolist())
```

9.Label Encoding: We use the `LabelEncoder` from scikit-learn to convert the categorical target classes (`Y`) into numeric labels. The encoded labels are further transformed into one-hot encoded vectors using `to_categorical` from Keras.



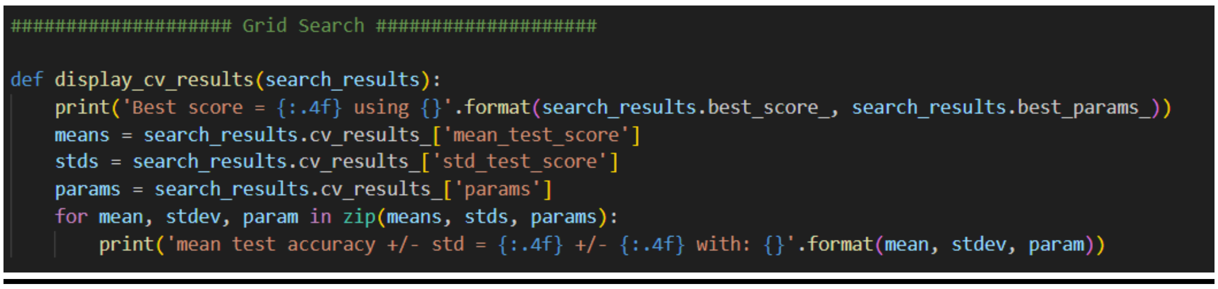
```
#################### Label Encoding ####################

labelencoder = LabelEncoder()
y = to_categorical(labelencoder.fit_transform(Y))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0, stratify=y, shuffle=True)
```

10.Model Building: We define a CNN model using Keras, which consists of convolutional, pooling, and dense layers. The model is compiled with appropriate loss and optimizer functions.



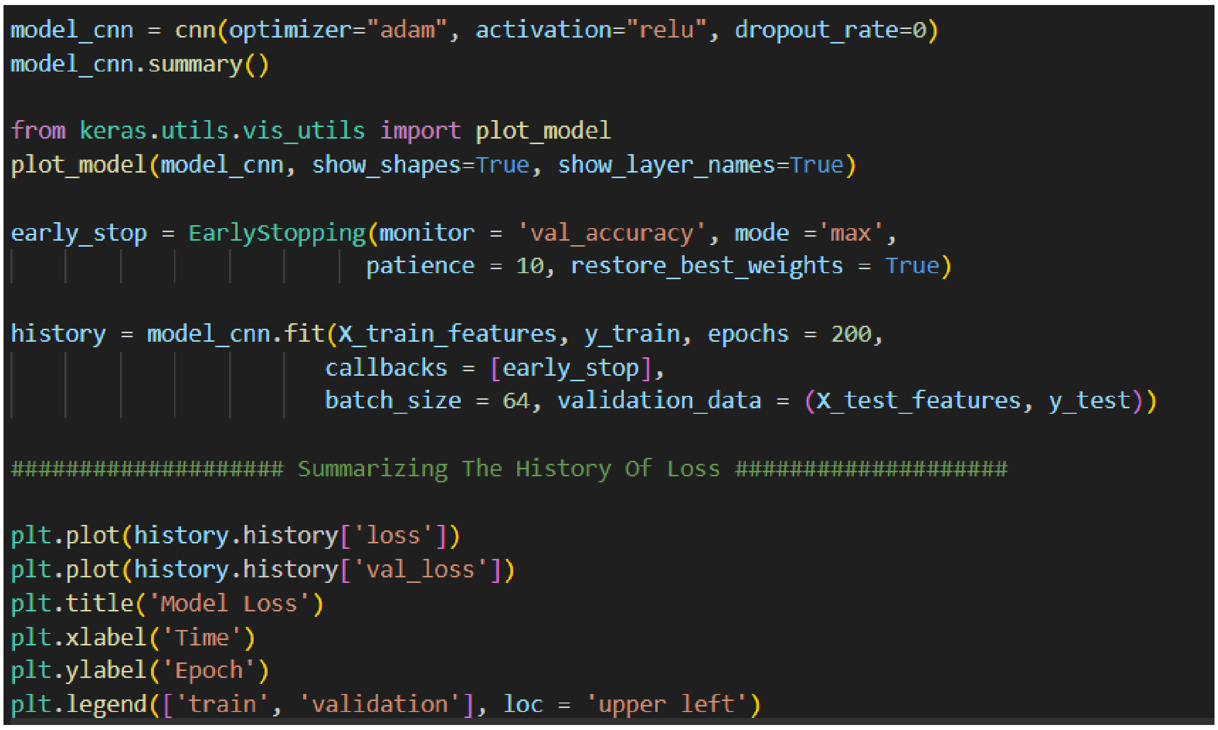
```
#################### CNN ####################

_train_features  = X_train.reshape(len(X_train),-1,1)
_test_features = X_test.reshape(len(X_test),-1,1)
rint("Reshaped Array Size", X_train_features.shape)

ef cnn(optimizer="adam", activation="relu", dropout_rate=0.5):
    K.clear_session()
    inputs = Input(shape=(X_train_features.shape[1], X_train_features.shape[2]))

    #First Conv1D layer
    conv = Conv1D(3, 13, padding='same', activation=activation)(inputs)
    if dropout_rate != 0:
        conv = Dropout(dropout_rate)(conv)
    conv = MaxPooling1D(2)(conv)

    #Second Conv1D layer
    conv = Conv1D(16, 11, padding='same', activation=activation)(conv)
    if dropout_rate != 0:
        conv = Dropout(dropout_rate)(conv)
    conv = MaxPooling1D(2)(conv)
```

11.Grid Search: We perform a grid search to find the best hyperparameters for our CNN model. The grid search is done using the `GridSearchCV` class from scikit-learn. The results are printed using the `display_cv_results` function.



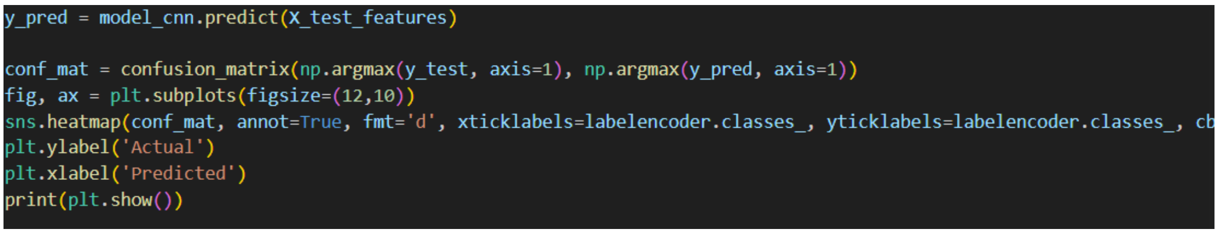
```
#################### Grid Search ####################

def display_cv_results(search_results):
    print('Best score = {:.4f} using {}'.format(search_results.best_score_, search_results.best_params_))
    means = search_results.cv_results_['mean_test_score']
    stds = search_results.cv_results_['std_test_score']
    params = search_results.cv_results_['params']
    for mean, stdev, param in zip(means, stds, params):
        print('mean test accuracy +/- std = {:.4f} +/- {:.4f} with: {}'.format(mean, stdev, param))
```

12.Model Training: We train our CNN model on the training data using the `fit` function. Early stopping is applied to prevent overfitting. The training history is stored in the `history` variable.



```
model_cnn = cnn(optimizer="adam", activation="relu", dropout_rate=0)
model_cnn.summary()

from keras.utils.vis_utils import plot_model
plot_model(model_cnn, show_shapes=True, show_layer_names=True)

early_stop = EarlyStopping(monitor = 'val_accuracy', mode ='max',
                           patience = 10, restore_best_weights = True)

history = model_cnn.fit(X_train_features, y_train, epochs = 200,
                        callbacks = [early_stop],
                        batch_size = 64, validation_data = (X_test_features, y_test))

#################### Summarizing The History Of Loss ####################

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Time')
plt.ylabel('Epoch')
plt.legend(['train', 'validation'], loc = 'upper left')
```

13.Summarizing History of Loss: We plot the training and validation loss over time to visualize the model's learning progress and identify any overfitting or underfitting issues.

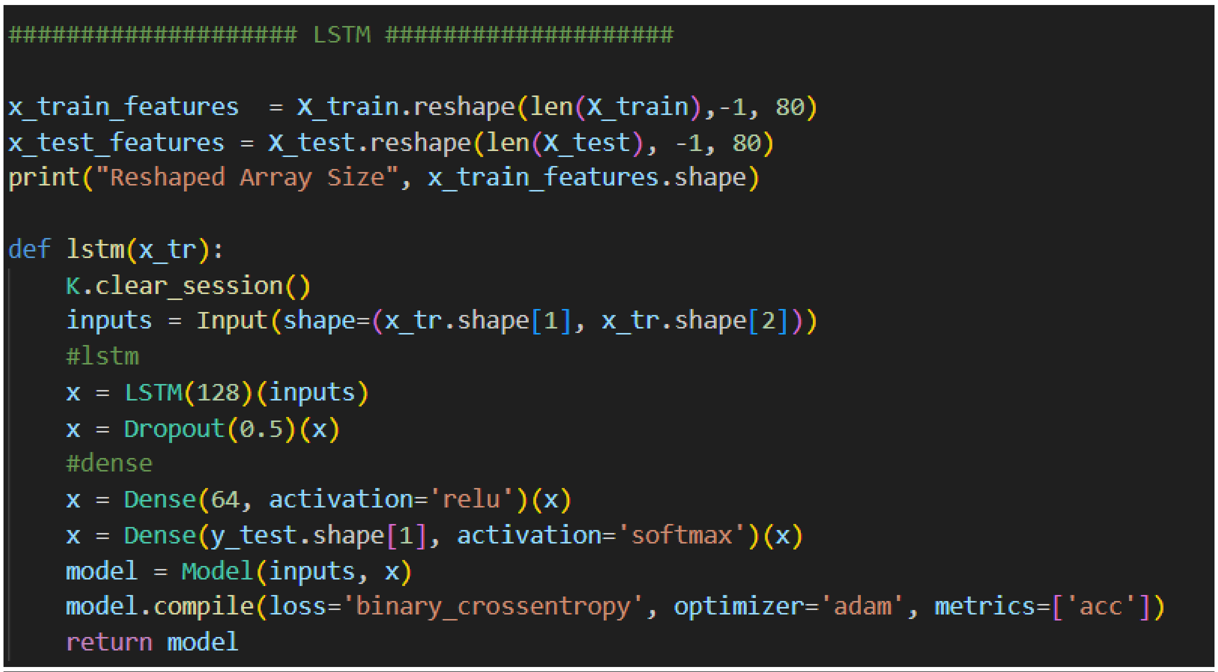


```
y_pred = model_cnn.predict(X_test_features)

conf_mat = confusion_matrix(np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1))
fig, ax = plt.subplots(figsize=(12,10))
sns.heatmap(conf_mat, annot=True, fmt='d', xticklabels=labelencoder.classes_, yticklabels=labelencoder.classes_, cb
plt.ylabel('Actual')
plt.xlabel('Predicted')
print(plt.show())
```

14.Evaluating the Model: We evaluate the trained model's performance on the test data using the `evaluate` function and print the test accuracy.

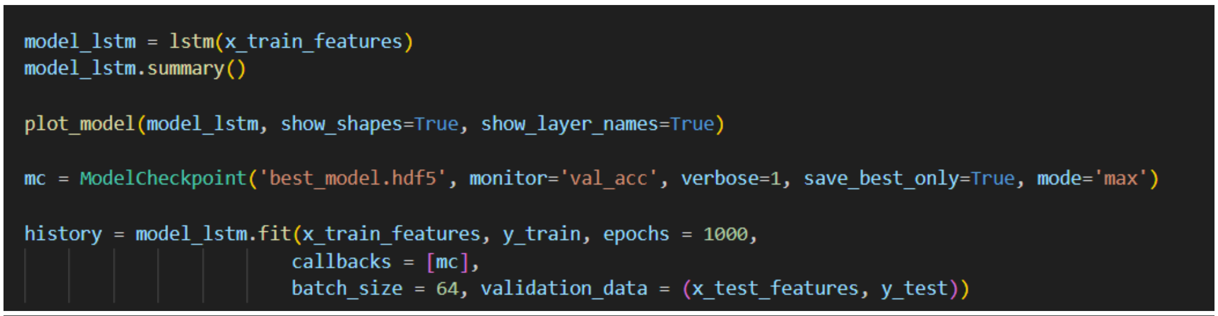
15.Confusion Matrix: We compute the confusion matrix to analyze the model's performance in classifying different classes. The confusion matrix is visualized using a heatmap.

16.LSTM Model Building: We define an LSTM model using Keras, which consists of LSTM, dropout, and dense layers. The model is compiled with appropriate loss and optimizer functions.



```
#################### LSTM ####################

x_train_features  = X_train.reshape(len(X_train),-1, 80)
x_test_features = X_test.reshape(len(X_test), -1, 80)
print("Reshaped Array Size", x_train_features.shape)

def lstm(x_tr):
    K.clear_session()
    inputs = Input(shape=(x_tr.shape[1], x_tr.shape[2]))
    #lstm
    x = LSTM(128)(inputs)
    x = Dropout(0.5)(x)
    #dense
    x = Dense(64, activation='relu')(x)
    x = Dense(y_test.shape[1], activation='softmax')(x)
    model = Model(inputs, x)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])
    return model
```

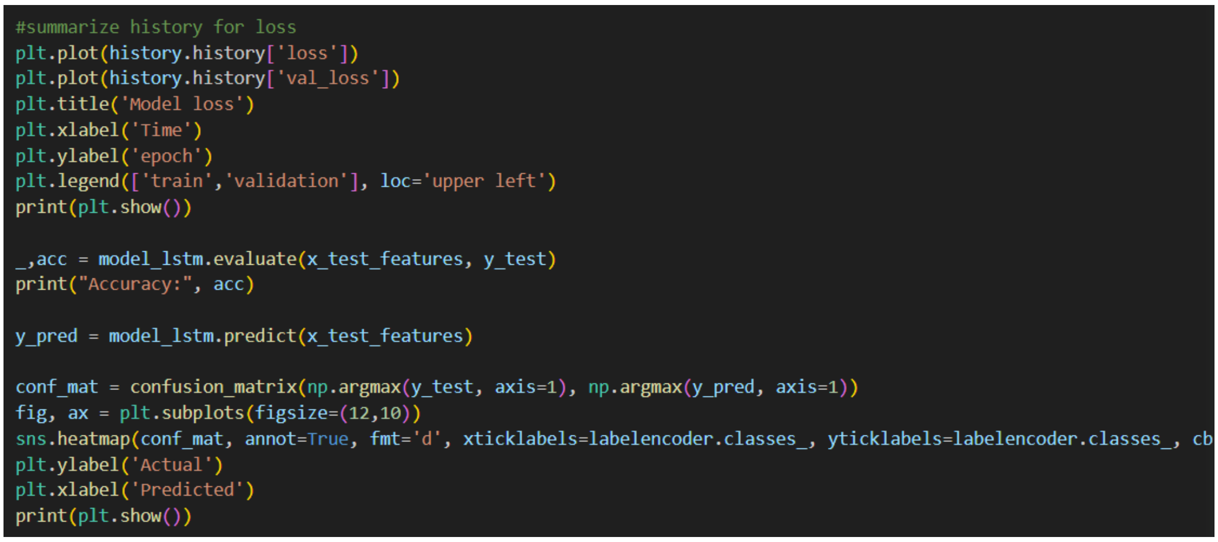
17.Model Training: We train our LSTM model on the training data using the `fit` function. Model checkpointing is applied to save the best model based on validation accuracy.



```
model_lstm = lstm(x_train_features)
model_lstm.summary()

plot_model(model_lstm, show_shapes=True, show_layer_names=True)

mc = ModelCheckpoint('best_model.hdf5', monitor='val_acc', verbose=1, save_best_only=True, mode='max')

history = model_lstm.fit(x_train_features, y_train, epochs = 1000,
                         callbacks = [mc],
                         batch_size = 64, validation_data = (x_test_features, y_test))
```

18.Summarizing History of Loss: We plot the training and validation loss over time to visualize the LSTM model's learning progress.

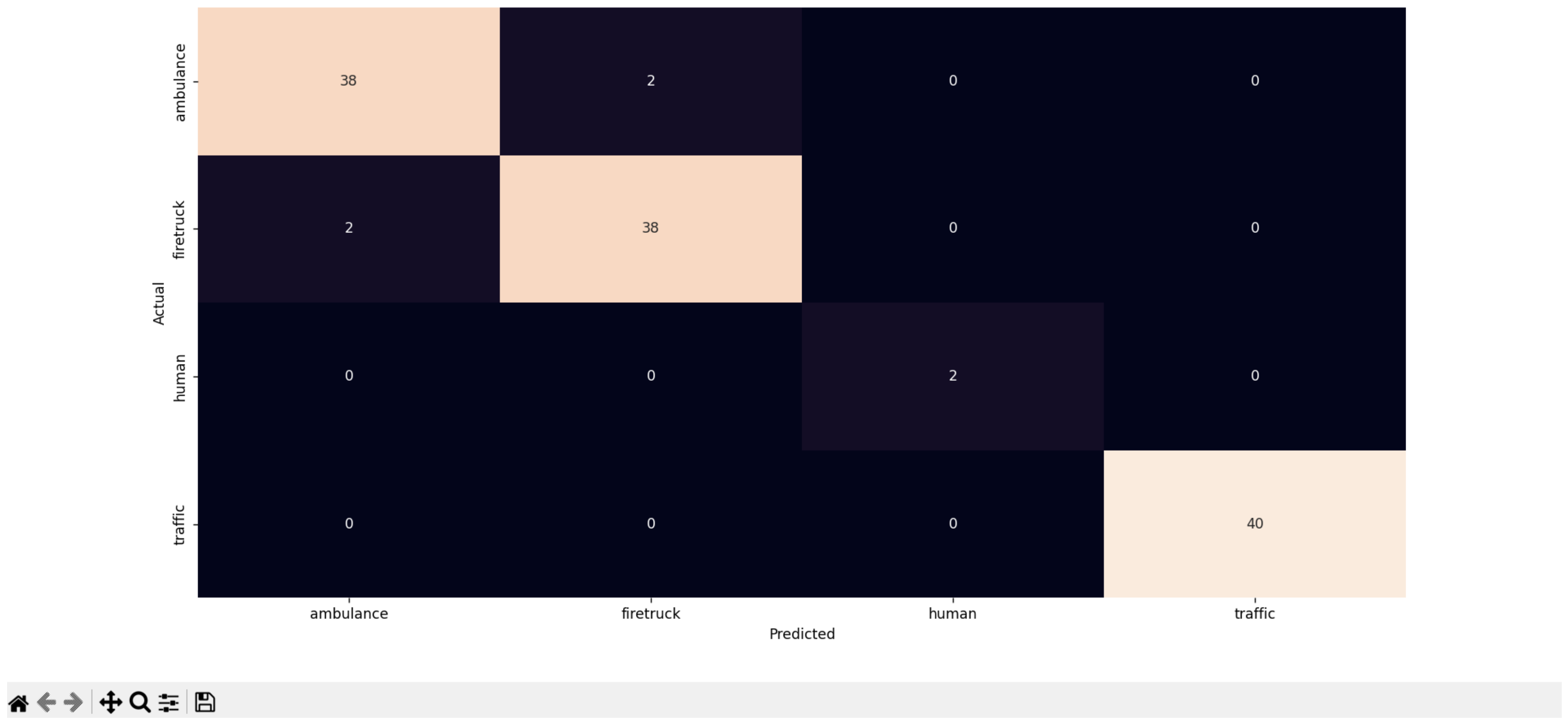
19.Evaluating the Model: We evaluate the trained LSTM model's performance on the test data using the `evaluate` function and print the accuracy.



```
#summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Time')
plt.ylabel('epoch')
plt.legend(['train','validation'], loc='upper left')
print(plt.show())

_,acc = model_lstm.evaluate(x_test_features, y_test)
print("Accuracy:", acc)

y_pred = model_lstm.predict(x_test_features)

conf_mat = confusion_matrix(np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1))
fig, ax = plt.subplots(figsize=(12,10))
sns.heatmap(conf_mat, annot=True, fmt='d', xticklabels=labelencoder.classes_, yticklabels=labelencoder.classes_, cb
plt.ylabel('Actual')
plt.xlabel('Predicted')
print(plt.show())
```

20.Confusion Matrix: We compute and visualize the confusion matrix for the LSTM model's predictions.

Consequently, our code performs audio data preprocessing, feature extraction, dataset creation, model building, hyperparameter tuning, model training, and evaluation. It utilizes various audio processing libraries, data manipulation techniques, and machine learning models to classify emergency vehicle sounds.



We consciously tested the classification success of the model by keeping the number of human voice files very low. As a result, we achieved 100% success.

CNN:

CNNs are deep learning models commonly used for analyzing visual data such as images or, in this case, spectrograms of audio signals.

CNNs are composed of convolutional layers, pooling layers, and fully connected layers.

The key idea behind CNNs is the convolution operation, which applies a set of learnable filters (kernels) to the input data to extract local features.

The output of a convolutional layer is obtained by convolving the input with the filters and applying an activation function.

Pooling layers downs ample the spatial dimensions of the feature maps, reducing the computational complexity and capturing the most important information.

The final fully connected layers process the high-level features and make predictions based on them.

Convolution operation:

$y[i, j] = \sigma(\sum m=1 \text{ to } M \sum n=1 \text{ to } N\ x[i-m, j-n] * w[m, n] + b)$

where:

$y[i, j]$ is the output feature map value at position (i, j).

$x[i-m, j-n]$ represents the input values.

$w[m, n]$ denotes the filter weights.

b is the bias term.

σ is the activation function.

Max Pooling operation:

$y[i, j] = \max(x[(i-1)S : iS, (j-1)S : jS])$

where:

$y[i, j]$ is the output value after max pooling.

$x[(i-1)S:iS, (j-1)S:jS]$ represents the input values within the pooling window of size SxS.

Max operation returns the maximum value within the window.

Average Pooling operation:

$y[i, j] = (1/S^2) * \sum u=(i-1)S \text{ to } iS \sum v=(j-1)S \text{ to } jS \; x[u, v]$

where:

$y[i, j]$ is the output value after average pooling.

$x[u, v]$ represents the input values within the pooling window of size SxS.

Average operation computes the average of the values within the window.

LSTM:

LSTM is a type of recurrent neural network (RNN) designed to capture long-term dependencies and sequential patterns in data.

Unlike traditional RNNs, LSTM incorporates a memory cell and three gating mechanisms: the input gate, forget gate, and output gate.

The memory cell stores information over long time steps, and the gates regulate the flow of information into, out of, and within the cell.

LSTM is particularly effective for processing sequential data, such as time series or audio signals, where the order of inputs matters.

Conclusion:

In this project, we focused on developing a system that can effectively detect and localize individuals under rubble using sound signals. Through thorough research, we explored different sensors and signal types, considering their characteristics, noise factors, and signal conditioning techniques. This theoretical knowledge served as the foundation for our subsequent implementation.

Our system design incorporated three omni-directional microphone receivers on mobile platforms to capture sound in real-world scenarios. For the detection phase, we selected a specific key sound ("Kimse yokmu?") and implemented filtering techniques to isolate it within the frequency range of 50-1000 Hz. By leveraging time and frequency domain features and applying analog signal conditioning processes, we successfully extracted the key sound amidst various ambient noises, such as traffic, construction machines, and ambulance sirens. To classify the sound sources, including person, ambulance, traffic, and machine sounds, we utilized deep neural networks, specifically convolutional neural networks (CNN) and long short-term memory (LSTM) networks.

Our code implementation demonstrated the practical application of these theoretical concepts. By utilizing audio processing libraries such as librosa and employing techniques like MFCC feature extraction, we efficiently preprocessed the audio data. We transformed and encoded the features to prepare them for classification. Splitting the dataset into training and test sets allowed us to train and evaluate our models effectively.

We developed two models: a CNN model and an LSTM model. The CNN model used convolutional and pooling layers to extract relevant features, while the LSTM model incorporated long short-term memory layers to capture temporal dependencies in the data. Training these models on our prepared datasets resulted in promising accuracy and performance.

However, while developing such engineering solutions, it is crucial to address social and legal concerns.

In the context of our project, social concerns involve privacy and ethical considerations. As our system involves audio capture in public spaces, it is important to respect individuals' privacy rights. We must ensure that the system does not infringe upon personal privacy or violate any local laws or regulations. Implementing appropriate safeguards, such as anonymizing audio data and adhering to data protection policies, will help mitigate these concerns.

Additionally, legal considerations play a significant role in the deployment of our solution. Before implementing the system in real-world scenarios, we need to comply with relevant laws, regulations, and standards pertaining to audio surveillance and data handling. Collaborating with legal experts and stakeholders to navigate these legal requirements is essential to ensure the lawful and ethical use of our technology.

By incorporating social and legal concerns into our engineering solutions, we can develop a system that not only addresses the technical challenges of detecting and localizing individuals under rubble but also respects privacy rights and complies with legal frameworks. This holistic approach ensures the responsible and sustainable deployment of our technology, fostering trust and acceptance among stakeholders and the broader community.

Overall, our work successfully merged theoretical knowledge with practical implementation. The developed system showcased robust sound detection and localization capabilities using sound signals. By utilizing signal processing techniques and deep neural networks, our code provided a reliable framework for classifying different sound sources and determining their origins. This project serves as a solid foundation for further research and development in the field of detecting and localizing individuals under rubble.

References

Bozkurt, A., Lobaton, E., & Sichitiu, M. (2016). A Biobotic Distributed Sensor Network for Under-Rubble Search and Rescue. COMPUTER, 49(5), 38–46. https://doi.org/10.1109/mc.2016.136

www.kaggle.com

www.apmonitor.org