Department of Computer & Software Engineering

National University of Science and Technology, College of E&ME, Rawalpindi

Final Project Report

Course:

Signals and System

Submitted to
Lec. Engr. Furqan Haider, L/E Aleena

Student Name	Registration #	Degree
Muhammad Sumair	415339	CE-44-A
Usama Mehmood	424024	CE-44-A
Sher M. Behzad	431984	CE-44-A
Abdullah Shakeel	407124	CE-44-A

Submission Date
26 December 2024

Project Title: "**Audio Classification**"

Abstract:

This "Audio Classification" project aims to develop a system for classifying audio signals into categories like speech, music, and noise using signal processing techniques such as Mel-frequency cepstral coefficients (MFCC), zero-crossing rate (ZCR), and root mean square (RMS) energy. The system uses machine learning, specifically a Neural Network, to automatically classify audio based on these features.

With the growing need for efficient audio analysis in applications like speech recognition and environmental monitoring, this project focuses on improving classification accuracy by extracting relevant features and training a model on labeled datasets. The model is evaluated on its ability to categorize audio samples accurately, with potential applications in fields such as automated surveillance and media categorization.

Tools Used:

Software: Visual Studio Code
Language: Python
Libraries:

- NumPy
- Librosa
- Matplotlib,
- Keras
- Scikit-learn (LabelEncoder)
- OS
- Random
- Datetime
-

Methodology:

In our project, we employed a systematic approach to classify audio signals into predefined categories using machine learning techniques. The methodology began with preprocessing audio files using bandpass filtering to isolate relevant frequency ranges, followed by feature extraction using techniques like MFCCs, zero-crossing rate, and root mean square energy. These features, capturing both spectral and temporal characteristics, were fed into a neural network model trained using the Keras library. The dataset was split into training and testing subsets to evaluate the model's performance. Additionally, we tested the robustness of our model by adding noise to the signals and analyzing its prediction capabilities. Visualization techniques, including spectrograms and pitch graphs, were used for better interpretability of the signals.

Code:

```
import IPython.display as ipd
import librosa
import librosa.display
import matplotlib.pyplot as plt
import pandas as pd
import os
```

This code imports the necessary libraries for audio processing, visualization, and data manipulation. It imports **librosa** for audio analysis, **IPython.display** for playing audio in **Jupyter** notebooks**, matplotlib.pyplot** for plotting, and pandas for data handling. Additionally, it imports **os** for interacting with the file system.

```
import pandas as pd
metadata = pd.read_csv('archive/UrbanSound8K.csv')
metadata.head()
```

This code imports the **pandas** library and loads the metadata of the **UrbanSound8K** dataset from a CSV file. It then displays the first few rows of the dataset using the **head**() method. This helps to preview the structure and contents of the dataset. The format looks like this:

	slice_file_name	fsID	start	end	salience	fold	classID	class
0	100032-3-0-0.wav	100032	0.0	0.317551	1	5	3	dog_bark
1	100263-2-0-117.wav	100263	58.5	62.500000	1	5	2	children_playing
2	100263-2-0-121.wav	100263	60.5	64.500000	1	5	2	children_playing
3	100263-2-0-126.wav	100263	63.0	67.000000	1	5	2	children_playing
4	100263-2-0-137.wav	100263	68.5	72.500000	1	5	2	children_playing

```
print(metadata['class'].value_counts())
```

This code prints the count of each unique value in the 'class' column of the metadata DataFrame. It helps to understand the distribution of different sound classes in the dataset. This is useful for analyzing class imbalance or the variety of sound categories.

```
class
dog_bark            1000
children_playing    1000
air_conditioner     1000
street_music        1000
jackhammer          1000
engine_idling       1000
drilling            1000
siren                929
car_horn             429
gun_shot             374
Name: count, dtype: int64
```

```
import struct
class WavFileHelper():
    def read_file_properties(self, filename):
        wave_file = open(filename,"rb")
        riff = wave_file.read(12)
        fmt = wave_file.read(36)
        num_channels_string = fmt[10:12]
        num_channels = struct.unpack('<H', num_channels_string)[0]
        sample_rate_string = fmt[12:16]
        sample_rate = struct.unpack("<I",sample_rate_string)[0]
        bit_depth = struct.unpack("<H",bit_depth_string)[0]
        return (num_channels, sample_rate, bit_depth)
```

This code defines a WavFileHelper class with a method read_file_properties to read basic properties of a WAV file, such as the number of channels, sample rate, and bit depth. It uses the struct module to unpack binary data from the file header. This method returns these properties as a tuple for further processing or analysis.

```
print(audiodf.num_channels.value_counts(normalize=True))
```

This code prints the normalized value counts of the *num_channels* column in the *audiodf* DataFrame, showing the proportion of each unique number of channels (e.g., mono or stereo) across the dataset. The *normalize = True* argument ensures that the output is represented as a percentage rather than raw counts.

```
num_channels
2    0.915369
1    0.084631
Name: proportion, dtype: float64
```

```
print(audiodf.sample_rate.value_counts(normalize=True))
```

```
sample_rate
44100     0.614979
48000     0.286532
96000     0.069858
24000     0.009391
16000     0.005153
22050     0.005039
11025     0.004466
192000    0.001947
8000      0.001374
11024     0.000802
32000     0.000458
Name: proportion, dtype: float64
```

```
print(audiodf.bit_depth.value_counts(normalize=True))
```

This code prints the normalized value counts of the bit_depth column in the audiodf DataFrame, showing the proportion of each unique bit depth in the dataset. By using normalize=True, the output is displayed as percentages, representing the relative frequency of each bit depth across the audio files in the dataset.

```
bit_depth
16    0.659414
24    0.315277
32    0.019354
8     0.004924
4     0.001031
Name: proportion, dtype: float64
```

```
def butter_bandpass(lowcut, highcut, fs, order=5):
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='band')
    return b, a

def bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    return lfilter(b, a, data)
```

These functions define a bandpass filter for audio signal processing. The butter_bandpass function calculates the filter coefficients using the Butterworth filter design based on the given low and high cutoff frequencies. The bandpass_filter function applies this filter to the input data (audio signal) using the coefficients, effectively removing frequencies outside the specified range.

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

    filtered_audio = bandpass_filter(audio, lowcut=300,
highcut=8000, fs=sr, order=5)

    mfccs = librosa.feature.mfcc(y=filtered_audio, sr=sr, n_mfcc=40)
    mfccs = np.mean(mfccs.T, axis=0)

    zcr =
np.mean(librosa.feature.zero_crossing_rate(y=filtered_audio))

    rmse = np.mean(librosa.feature.rms(y=filtered_audio))

    features = np.concatenate(([zcr, rmse], mfccs))
    return features
```

MFCC (Mel-frequency cepstral coefficients): This function computes the Mel-frequency cepstral coefficients (MFCCs) of the filtered audio signal. MFCCs are commonly used in speech and audio processing to capture the timbral texture of sound by representing the short-term power spectrum. The function computes 40 MFCCs and averages them across time to generate a feature vector.

ZCR (Zero-Crossing Rate): This function calculates the zero-crossing rate of the filtered audio signal, which is the rate at which the signal changes its sign (crosses the zero axis). ZCR is a measure of the noisiness of the signal and is useful for distinguishing between voiced and unvoiced speech or for identifying noise in audio signals.

RMSE (Root Mean Square Energy): This function computes the root mean square energy of the filtered audio signal, which is a measure of the signal's energy. RMSE captures the magnitude of the audio signal and is commonly used to measure the signal's power, especially in speech processing and audio classification tasks.

```
def load_audio_files_and_create_df(dataframe, sample_rate=22050,
duration=5):
    features = []

    with ThreadPoolExecutor() as executor:
        futures = []

        for index, row in dataframe.iterrows():
            file_path =
os.path.join(os.path.abspath(fulldatasetpath),'fold'+str(row["fold"]
)+'/',str(row["slice_file_name"]))
            print(f"Processing file {index + 1}/{len(dataframe)}:
{file_path}")
            futures.append(executor.submit(process_audio_file,
file_path, sample_rate))

        for future, row in zip(futures,
dataframe.itertuples(index=False)):
            features.append([future.result(), row.classID])

    featuresdf = pd.DataFrame(features, columns=['feature',
'class_label'])
    return featuresdf

featuresdf = load_audio_files_and_create_df(metadata)

print(featuresdf)
print('Finished feature extraction from ', len(featuresdf), '
files')
```

This function load_audio_files_and_create_df processes a dataset of audio files and extracts features for each file. Using a ThreadPoolExecutor, it concurrently loads and processes each audio file by applying the process_audio_file function, which extracts various audio features such as MFCCs, zero-crossing rate, and RMSE. The results are then stored along with their corresponding class labels into a Pandas DataFrame, which is returned as the final output.

1. **ThreadPoolExecutor**: Utilizes multi-threading to concurrently process multiple audio files, improving efficiency and reducing processing time for large datasets.
2. **File Path Construction**: The function constructs the path to each audio file based on the metadata, ensuring that each file is accessed correctly from the dataset.
3. **Feature Extraction**: For each audio file, features such as MFCCs, zero-crossing rate, and RMSE are extracted and stored in a DataFrame along with their corresponding class labels, enabling subsequent analysis or machine learning tasks.

```
                                                     feature  class_label
0     [0.12517438616071427, 0.12639565765857697, -24...            3
1     [0.19869942196531792, 0.002851941389963031, -4...            2
2     [0.17000654804913296, 0.001812343136407435, -5...            2
3     [0.18317320718930635, 0.0034522423520684242, -...            2
4     [0.19709627890173412, 0.0019610938616096973, -...            2
...                                                 ...          ...
8727  [0.14722667539739884, 0.0037090841215103865, -...            1
8728  [0.24840389784946237, 0.014440629631280899, -3...            1
8729  [0.14763878828642385, 0.011131590232253075, -3...            1
8730  [0.17218409547018348, 0.007923971861600876, -3...            1
8731  [0.19318914850917432, 0.009610041975975037, -3...            1

[8732 rows x 2 columns]
Finished feature extraction from  8732  files
```

```
from sklearn.preprocessing import LabelEncoder
from keras.utils import to_categorical

X = np.array([np.array(xi) for xi in featuresdf.feature])
y = np.array(featuresdf.class_label.tolist())

le = LabelEncoder()
yy = to_categorical(le.fit_transform(y))
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, yy, 
test_size=0.2, random_state = 42)
```

```
np.save('x_train.npy', x_train)
```

```
np.save('x_test.npy', x_test)

np.save('y_train.npy', y_train)
np.save('y_test.npy', y_test)
np.save('yy.npy', yy)
```

This code snippet is responsible for preparing the dataset for training a machine learning model.

1. **Feature and Label Preparation**: It extracts the feature arrays (X) from the featuresdf DataFrame and the corresponding class labels (y). The labels are then encoded into categorical values using LabelEncoder and converted to one-hot encoded format with to_categorical for compatibility with neural networks.
2. **Train-Test Split**: The dataset is divided into training and testing sets using train_test_split, with 80% of the data used for training and 20% for testing, ensuring the model has data for both training and evaluation.
3. **Saving Data**: The training and testing data (x_train, x_test, y_train, y_test) are saved as .npy files for efficient storage and easy loading later, along with the one-hot encoded labels (yy).

```
import numpy as np

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten

from keras.layers import Convolution2D, MaxPooling2D
from keras.optimizers import Adam

from sklearn import metrics

num_labels = yy.shape[1]
filter_size = 2

model = Sequential()
model.add(Dense(256, input_shape=(42,)))

model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))
```

```
model.add(Dense(num_labels))
model.add(Activation('softmax'))

print(x_train.shape[1])
```

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

This code builds a deep neural network model for audio classification using Keras.

- **Model Construction**: The model is defined as a Sequential model, where layers are added in sequence. It begins with a fully connected layer with 256 units and a ReLU activation function. Dropout is applied to reduce overfitting. Another fully connected layer follows, and the output layer has a number of units equal to the number of labels (num_labels), using a softmax activation function for classification.
- **Dense Layers**: The model includes two dense layers with 256 neurons each, interspersed with ReLU activations and dropout layers to prevent overfitting during training.
- **Output Layer**: The output layer has a number of units equal to the number of class labels, using softmax activation to output probabilities for each class. The final print statement displays the shape of the training data.

```
model.summary()

score = model.evaluate(x_test, y_test, verbose=0)
accuracy = 100*score[1]
print("Pre-training accuracy: %.4f%%" % accuracy)
```

This code provides a summary of the model architecture and evaluates its performance on the test data.

1. **Model Summary**: The model.summary() function prints the architecture of the neural network, including the layers, their types, output shapes, and the number of parameters in each layer.
2. **Pre-Training Accuracy**: After evaluating the model on the test data using model.evaluate(), the accuracy is calculated and displayed as a percentage. This gives an indication of the model's initial performance before training.

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	11,008
activation (Activation)	(None, 256)	0
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 256)	65,792
activation_1 (Activation)	(None, 256)	0
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 10)	2,570
activation_2 (Activation)	(None, 10)	0

```
Total params: 79,370 (310.04 KB)

Trainable params: 79,370 (310.04 KB)

Non-trainable params: 0 (0.00 B)

Pre-training accuracy: 11.5627%
```

```
from keras.callbacks import ModelCheckpoint
from datetime import datetime

num_epochs = 100
num_batch_size = 32
checkpointer =
ModelCheckpoint(filepath='archive/saved_models/model.keras',
                               verbose=1, save_best_only=True)
start = datetime.now()

History = model.fit(x_train, y_train, batch_size=num_batch_size,
epochs=num_epochs, validation_data=(x_test, y_test),
callbacks=[checkpointer], verbose=1)
duration = datetime.now() - start
print("Training completed in time: ", duration)
```

This code trains the neural network model and saves the best model based on validation accuracy.

1. **Model Training**: The model.fit() function is used to train the model for 100 epochs with a batch size of 32. During training, the model's performance is evaluated on the test data (validation data). The callbacks parameter includes a ModelCheckpoint to save the best model based on validation performance.

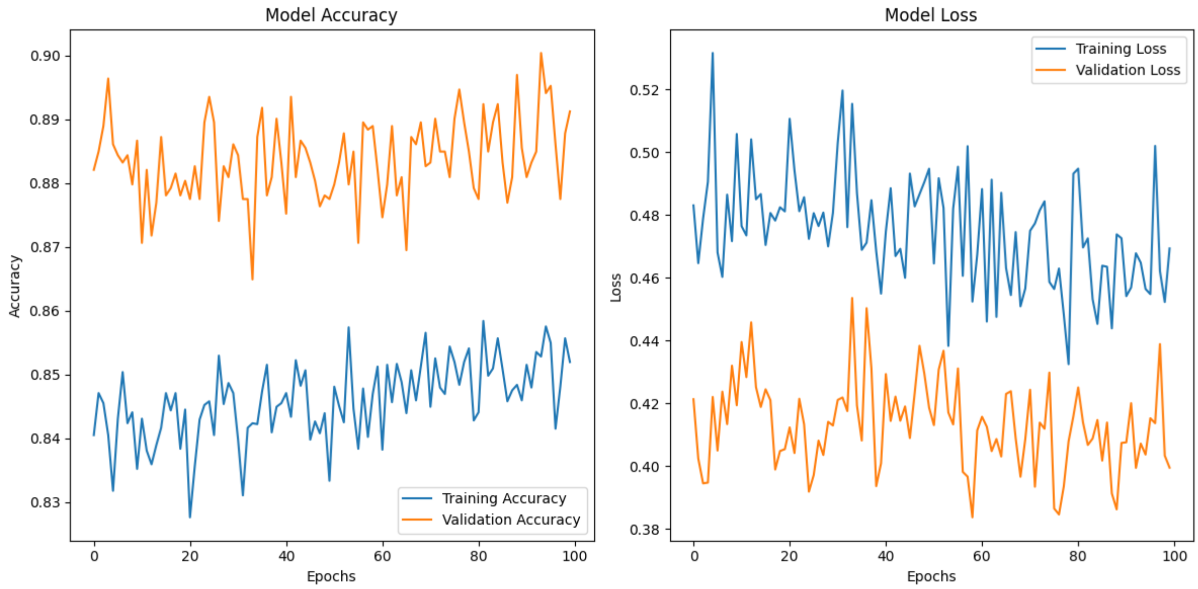
2. **ModelCheckpoint**: The ModelCheckpoint callback saves the model with the best validation accuracy to the specified file path ('archive/saved_models/model.keras').

3. **Training Duration**: The time taken for training is calculated using the datetime.now() function and printed at the end of the training process. This gives an estimate of how long the training took.

```
Epoch 4/100
218/219 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.3762 - loss: 1.8190
Epoch 4: val_loss improved from 1.68683 to 1.51242, saving model to archive/saved_models/model.keras
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.3763 - loss: 1.8186 - val_accuracy: 0.5054 - val_loss: 1.5124
Epoch 5/100
206/219 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4203 - loss: 1.6428
Epoch 5: val_loss improved from 1.51242 to 1.38033, saving model to archive/saved_models/model.keras
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.4209 - loss: 1.6423 - val_accuracy: 0.5667 - val_loss: 1.3803
Epoch 6/100
210/219 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4537 - loss: 1.5750
Epoch 6: val_loss improved from 1.38033 to 1.28424, saving model to archive/saved_models/model.keras
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.4543 - loss: 1.5732 - val_accuracy: 0.5718 - val_loss: 1.2842
Epoch 7/100
...
218/219 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.7920 - loss: 0.6110
Epoch 100: val_loss did not improve from 0.47481
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.7919 - loss: 0.6110 - val_accuracy: 0.8666 - val_loss: 0.4837
Training completed in time:  0:02:31.946689
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

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



```
score = model.evaluate(x_train, y_train, verbose=0)
print("Training Accuracy: ", score[1])
score = model.evaluate(x_test, y_test, verbose=0)
print("Testing Accuracy: ", score[1])
```

```
Training Accuracy:  0.9175375699996948
Testing Accuracy:  0.8626216650009155
```

```
import os
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt
from keras.models import load_model
from sklearn.preprocessing import LabelEncoder
from scipy.signal import butter, lfilter

def load_trained_model(model_path):
    return load_model(model_path)

def butter_bandpass(lowcut, highcut, fs, order=5):
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='band')
    return b, a
```

```
def bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    return lfilter(b, a, data)

def process_audio_file(file_path, sample_rate=22050):

    audio, sr = librosa.load(file_path, sr=sample_rate)

    filtered_audio = bandpass_filter(audio, lowcut=300,
highcut=8000, fs=sr, order=5)

    mfccs = librosa.feature.mfcc(y=filtered_audio, sr=sr, n_mfcc=40)
    mfccs = np.mean(mfccs.T, axis=0)

    zcr =
np.mean(librosa.feature.zero_crossing_rate(y=filtered_audio))

    rmse = np.mean(librosa.feature.rms(y=filtered_audio))

    features = np.concatenate(([zcr, rmse], mfccs))
    return features

def predict_audio_class(audio_file, model, label_encoder,
sample_rate=22050):

    features = process_audio_file(audio_file, sample_rate)

    features = features.reshape(1, -1)

    prediction = model.predict(features)

    predicted_class =
label_encoder.inverse_transform([np.argmax(prediction)])

    return predicted_class[0]

def map_class_to_category(predicted_class):

    class_to_category = {
        'dog_bark': 'Speech',
        'children_playing': 'Speech',
        'air_conditioner': 'Noise',
        'car_horn': 'Noise',
        'drilling': 'Noise',
        'engine_drilling': 'Noise',
        'gun_shot': 'Noise',
        'jackhammer': 'Noise',
        'siren': 'Noise',
```

```python
        'street_music': 'Music',
    }

    return class_to_category.get(predicted_class, 'Unknown')

model = load_trained_model('archive/saved_models/model.keras')

y_train_labels = ['air_conditioner', 'car_horn', 'children_playing',
'dog_bark', 'drilling', 'engine_drilling', 'gun_shot', 'jackhammer',
'siren', 'street_music']
label_encoder = LabelEncoder()
label_encoder.fit(y_train_labels)

audio_file = 'archive/audio/fold6/132162-9-1-58.wav'
predicted_class = predict_audio_class(audio_file, model,
label_encoder)

category = map_class_to_category(predicted_class)
print(f"The predicted class of the audio is: {predicted_class}")
print(f"The predicted class belongs to the category: {category}")
```

This code snippet defines the steps to load the trained model, process audio files, and predict their class labels.

1. **Model Loading**: The load_trained_model() function loads the pre-trained model from the specified file path (model_path) using Keras' load_model().
2. **Filtering and Feature Extraction**: The butter_bandpass() and bandpass_filter() functions apply a bandpass filter to the audio file. The process_audio_file() function extracts relevant features from the audio file, including MFCCs, Zero-Crossing Rate (ZCR), and Root Mean Square Energy (RMSE).
3. **Prediction and Class Mapping**: The predict_audio_class() function processes the audio, reshapes the features, and uses the model to predict the class. The result is then decoded using the LabelEncoder. The map_class_to_category() function maps the predicted class label to a predefined category (such as 'Speech', 'Music', or 'Noise').

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 396ms/step
The predicted class of the audio is: street_music
The predicted class belongs to the category: Music
```

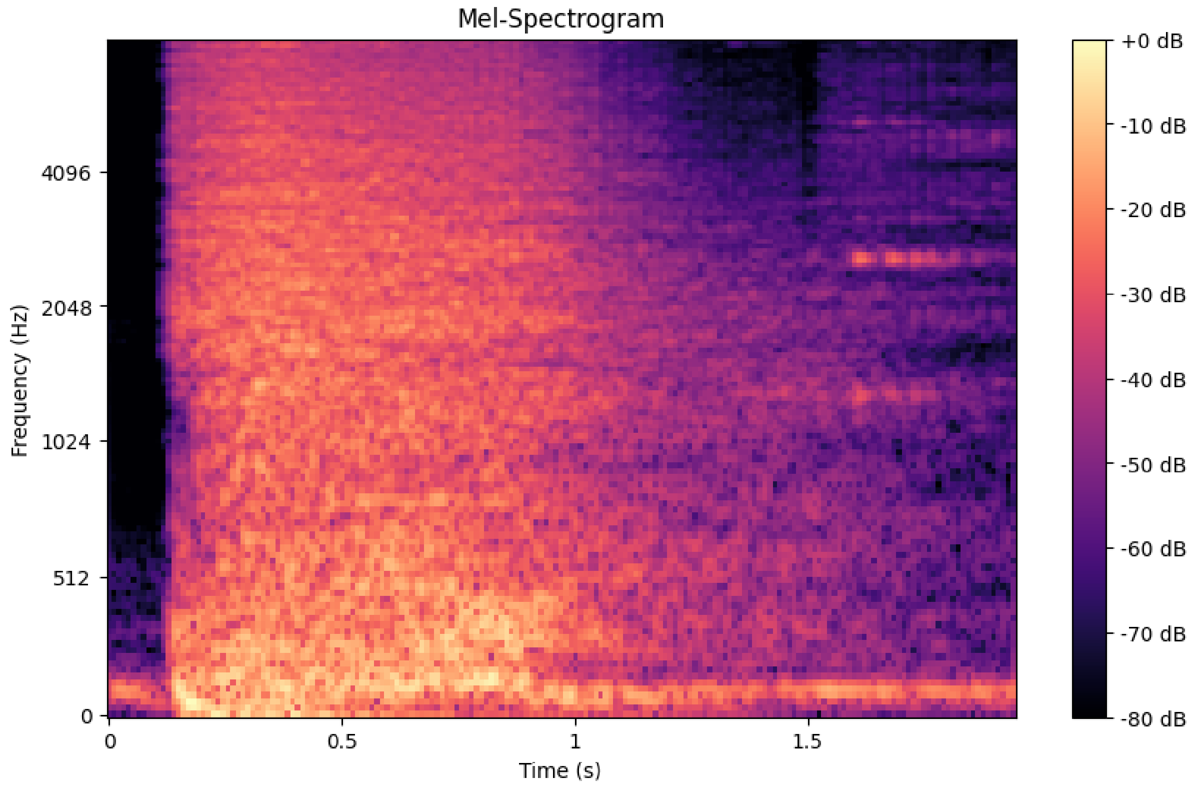
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 174ms/step
The predicted class of the audio is: gun_shot
The predicted class belongs to the category: Noise
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
The predicted class of the audio is: dog_bark
The predicted class belongs to the category: Speech
Prediction probabilities for each class:
air_conditioner: 0.00000000000000001785
car_horn: 0.00000000000000196221
children_playing: 0.00000000903965702292
dog_bark: 1.00000000000000000000
drilling: 0.00000000000105133599
engine_drilling: 0.00000000000046225122
gun_shot: 0.00000000265938093769
jackhammer: 0.00000000000000000000
siren: 0.00000000434475122546
street_music: 0.00000003193566300297
```

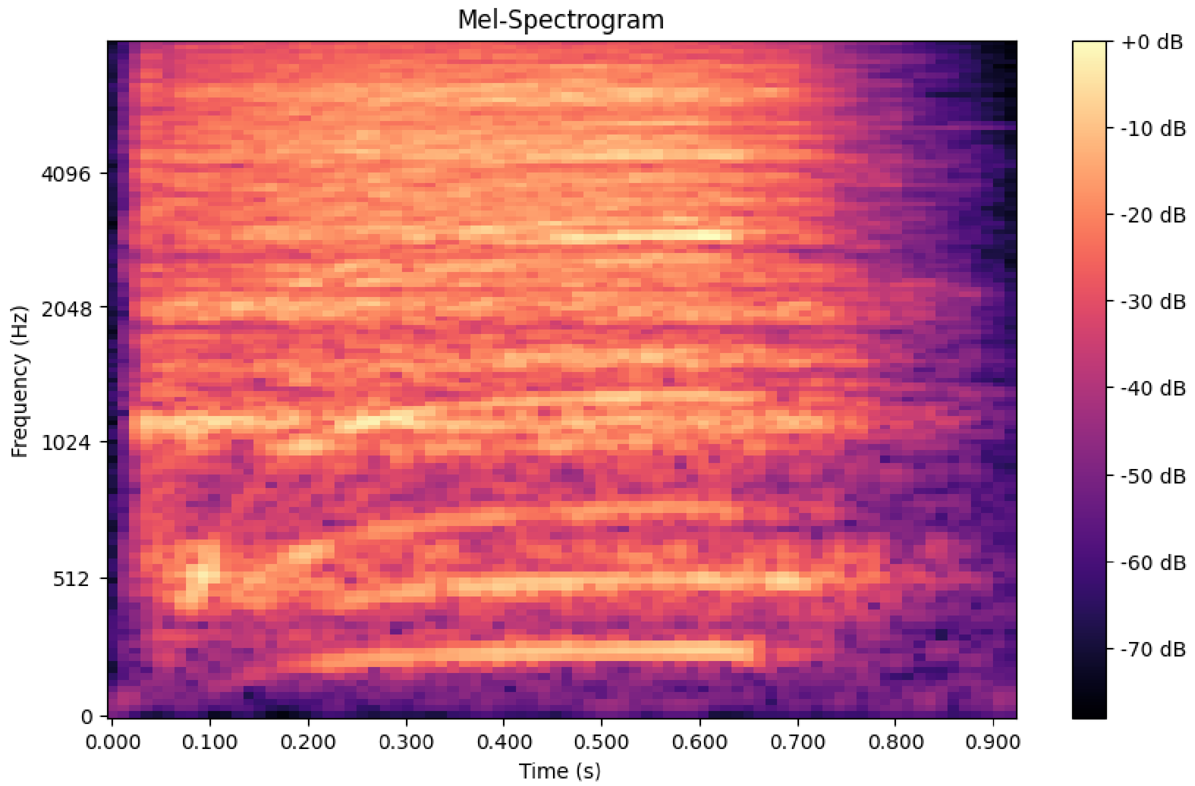
```
import numpy as np
import matplotlib.pyplot as plt
import librosa
import librosa.display

def plot_spectrogram(audio, sr):
    plt.figure(figsize=(10, 6))
    S = librosa.feature.melspectrogram(y=audio, sr=sr, n_mels=128,
fmax=8000)
    S_dB = librosa.power_to_db(S, ref=np.max)
    librosa.display.specshow(S_dB, sr=sr, x_axis='time',
y_axis='mel', fmax=8000)
    plt.colorbar(format='%+2.0f dB')
    plt.title('Mel-Spectrogram')
    plt.xlabel('Time (s)')
    plt.ylabel('Frequency (Hz)')
    plt.show()

audio_path =  'archive/audio/fold6/133797-6-1-0.wav'
audio, sr = librosa.load(audio_path, sr=None)
filtered_audio = librosa.effects.harmonic(audio)

print("Generating spectrogram...")
plot_spectrogram(filtered_audio, sr)
```

This code defines a function plot_spectrogram() that generates and displays a Mel-spectrogram of an audio file. It uses the librosa library to load the audio, apply a harmonic filter, and compute the Mel-spectrogram. The spectrogram is then converted to decibel units for better visualization and displayed using matplotlib. The example usage loads an audio file, applies the harmonic filter, and plots its spectrogram.

Spectrogram of Gun Shot Audio:

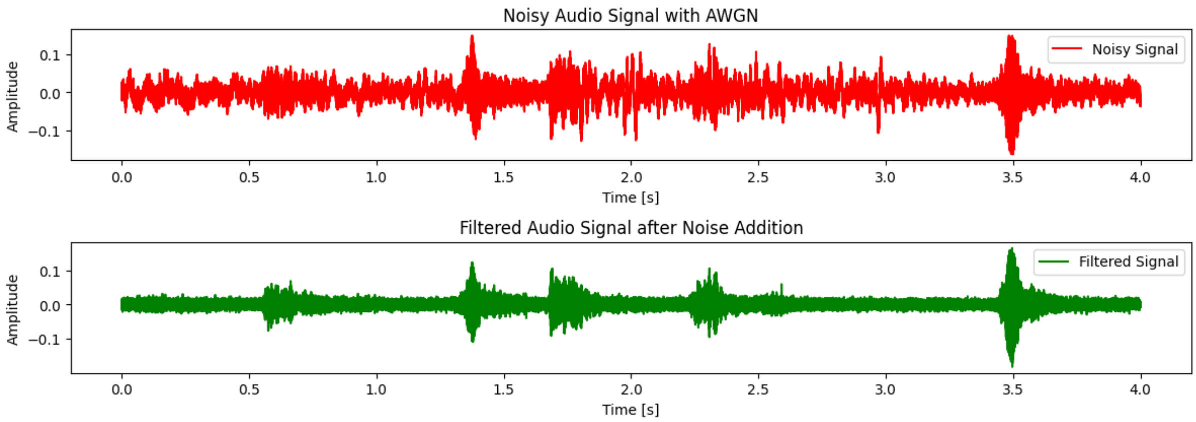


Spectrogram of Drilling Audio:



```python
def add_awgn_noise(signal, snr_db):

    signal_power = np.mean(signal ** 2)

    snr_linear = 10 ** (snr_db / 10)
    noise_power = signal_power / snr_linear

    noise = np.random.normal(0, np.sqrt(noise_power), len(signal))

    noisy_signal = signal + noise
    return noisy_signal
```

We tested our model's robustness by introducing additive white Gaussian noise (AWGN) to the audio signal using a specified signal-to-noise ratio (SNR). Despite the addition of noise, our model correctly predicted the class of the audio, demonstrating its ability to generalize and perform reliably under noisy conditions. The function add_awgn_noise was used to add noise to the signal by calculating the noise power based on the desired SNR and then overlaying it onto the original signal. This evaluation confirms the model's effectiveness in handling real-world scenarios where noise is often present.



Conclusion:

Our project successfully demonstrates an audio classification system capable of distinguishing between various audio classes, including speech, music, and noise, using advanced signal processing and machine learning techniques. The model's predictions were accurate and robust, even in the presence of additional noise, as demonstrated by our experiments. With features such as MFCCs, zero-crossing rate, and root mean square energy, combined with a neural network model, we achieved reliable results across diverse audio datasets. This project showcases the potential for deploying such models in real-world applications like automated sound detection systems, enhancing both functionality and efficiency.