```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!pip install pydub python_speech_features
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting pydub
  Downloading pydub-0.25.1-py2.py3-none-any.whl (32 kB)
Collecting python_speech_features
  Downloading python_speech_features-0.6.tar.gz (5.6 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: python_speech_features
  Building wheel for python_speech_features (setup.py) ... done
  Created wheel for python_speech_features: filename=python_speech_features-0.6-py3-none-any.whl size=5886 sha256=2726ece95ff901b839d16
  Stored in directory: /root/.cache/pip/wheels/09/a1/04/08e2688d2562d8f9ff89e77c6ddfbf7268e07dae1a6f22455e
Successfully built python_speech_features
Installing collected packages: python_speech_features, pydub
Successfully installed pydub-0.25.1 python_speech_features-0.6
```

Importing Packages

```
import os
import pydub
from pydub import AudioSegment
from pydub.silence import split_on_silence
from python_speech_features import mfcc
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from keras.utils import to_categorical
from sklearn.metrics import classification_report
import pandas as pd
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from sklearn.model_selection import train_test_split
from keras.models import load_model
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import confusion_matrix
import warnings
warnings.filterwarnings('ignore')
```

Data Preprocessing

```
# Set directory path
directory = '/content/drive/MyDrive/audio/'

# Get list of all folders in directory
class_labels = [folder for folder in os.listdir(directory) if os.path.isdir(os.path.join(directory, folder))]

# Print list of folders
print(class_labels)
```

```
['.ipynb_checkpoints', 'Siren', 'Dog_bark', 'Car_honk']
```

```
# Set parameters
n_mfcc = 20

# Initialize lists for audio data and labels
X = []
y = []

# Loop through each class directory
for i, label in enumerate(class_labels):
    class_dir = f'/content/drive/MyDrive/audio/{label}/'
```

```
    # Loop through each audio file in the class directory
    for filename in os.listdir(class_dir):
        file_path = os.path.join(class_dir, filename)

        try:
            # Load audio file
            audio = AudioSegment.from_file(file_path)
            audio_data = np.array(audio.get_array_of_samples())
            sample_rate = audio.frame_rate

            # Split audio into chunks of silence
            audio_chunks = split_on_silence(audio, min_silence_len=500, silence_thresh=-50)

            # Extract MFCCs from each chunk of audio
            nfft = 2400 # Set larger FFT size
            for chunk in audio_chunks:
                chunk_data = np.array(chunk.get_array_of_samples())
                mfccs = mfcc(chunk_data, samplerate=sample_rate, numcep=n_mfcc, nfft=nfft) # Set larger nfft value
                mfccs = np.mean(mfccs, axis=0)

                # Append MFCCs and label to lists
                X.append(mfccs)
                y.append(i)

        except Exception as e:
            # Skip file if it is not a recognized audio format
            print(f'Skipping file {filename}: {e}')
```

```
Skipping file 421017__girlwithsoundrecorder__ambulance-in-oradea.wav: 64
WARNING:root:frame length (4800) is greater than FFT size (2400), frame will be truncated. Increase NFFT to avoid.
WARNING:root:frame length (4800) is greater than FFT size (2400), frame will be truncated. Increase NFFT to avoid.
WARNING:root:frame length (4800) is greater than FFT size (2400), frame will be truncated. Increase NFFT to avoid.
WARNING:root:frame length (4800) is greater than FFT size (2400), frame will be truncated. Increase NFFT to avoid.
```

```
X
```

```
       array([ 15.80865061,  -9.0934399 ,  -6.20958822,  -0.25560043,
              -13.40476851,   3.9670414 , -16.04360961, -19.35241739,
              -16.56447103,  -0.20680604,  -4.88178869, -11.38483015,
              -19.14385198,   0.47698549,  -1.21456369,   4.50848918,
               -2.07203365,   4.80494218,   3.43553672,   2.16421708]),
       array([ 16.15650001,  -8.63001985,  -6.53428947,  -0.38722278,
              -14.32747198,   4.66663939, -17.71296252, -20.87732943,
              -17.74653764,  -0.32977062,  -4.48167561, -11.47103992,
              -19.9104495 ,  -0.17954367,  -1.9645522 ,   4.86571316,
               -2.65490904,   4.81955943,   3.54586712,   1.97988416]),
       array([ 16.50945607,  -9.94395702,  -5.16315747,  -0.81683667,
              -16.26680066,   5.79466111, -15.60019426, -18.80002058,
              -16.27759185,  -1.52753722,  -5.71379585, -10.69927539,
              -19.78091492,   0.52216258,  -0.27978473,   5.90055841,
               -2.02855947,   3.25815419,   3.63385809,   2.23112649])]
```

```
y
```

```
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3,
       3]
```

```
# Convert labels to one-hot encoded vectors
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

onehot_encoder = OneHotEncoder(sparse=False)
y = onehot_encoder.fit_transform(y.reshape(len(y), 1))
```

```
y
```

```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       ...,
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.]])
```

```
# Convert lists to numpy arrays
X = np.array(X)
y = np.array(y)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
X_train.shape
```

```
(630, 20)
```

Model Training and Testing

```
# Define the model architecture
model = Sequential()
model.add(Dense(256, input_shape=(X.shape[1],)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(3))
model.add(Activation('softmax'))

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

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               5376

 activation (Activation)     (None, 256)               0

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 128)               32896

 activation_1 (Activation)   (None, 128)               0

 dropout_1 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 3)                 387

 activation_2 (Activation)   (None, 3)                 0

=================================================================
Total params: 38,659
Trainable params: 38,659
Non-trainable params: 0
_________________________________________________________________
```

```
# Train the model
history = model.fit(X_train, y_train,
                    batch_size=32,
                    epochs=100,
                    validation_data=(X_test, y_test))
```

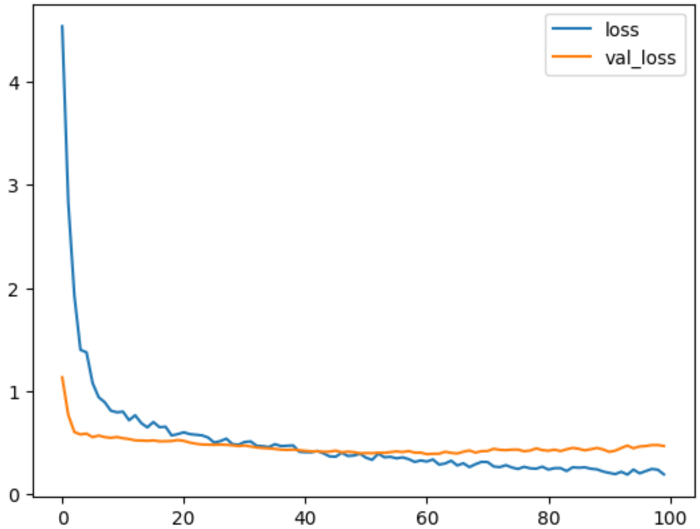
```
Epoch 73/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2637 - accuracy: 0.8968 - val_loss: 0.4316 - val_accuracy: 0.8963
Epoch 74/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2829 - accuracy: 0.8921 - val_loss: 0.4284 - val_accuracy: 0.9037
Epoch 75/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2602 - accuracy: 0.9143 - val_loss: 0.4333 - val_accuracy: 0.8963
Epoch 76/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2463 - accuracy: 0.9143 - val_loss: 0.4336 - val_accuracy: 0.9074
Epoch 77/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2669 - accuracy: 0.9032 - val_loss: 0.4177 - val_accuracy: 0.9037
Epoch 78/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2523 - accuracy: 0.9016 - val_loss: 0.4244 - val_accuracy: 0.9000
Epoch 79/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2485 - accuracy: 0.9079 - val_loss: 0.4455 - val_accuracy: 0.8963
Epoch 80/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2682 - accuracy: 0.9032 - val_loss: 0.4290 - val_accuracy: 0.8926
Epoch 81/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2400 - accuracy: 0.9143 - val_loss: 0.4228 - val_accuracy: 0.9074
Epoch 82/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2540 - accuracy: 0.8984 - val_loss: 0.4333 - val_accuracy: 0.9037
Epoch 83/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2543 - accuracy: 0.9095 - val_loss: 0.4195 - val_accuracy: 0.9037
Epoch 84/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2262 - accuracy: 0.9159 - val_loss: 0.4368 - val_accuracy: 0.9000
Epoch 85/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2622 - accuracy: 0.8921 - val_loss: 0.4504 - val_accuracy: 0.9148
Epoch 86/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2582 - accuracy: 0.8905 - val_loss: 0.4414 - val_accuracy: 0.9000
Epoch 87/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2619 - accuracy: 0.9016 - val_loss: 0.4269 - val_accuracy: 0.9000
Epoch 88/100
20/20 [==============================] - 0s 6ms/step - loss: 0.2483 - accuracy: 0.9095 - val_loss: 0.4376 - val_accuracy: 0.8852
Epoch 89/100
20/20 [==============================] - 0s 5ms/step - loss: 0.2422 - accuracy: 0.9063 - val_loss: 0.4501 - val_accuracy: 0.8963
Epoch 90/100
20/20 [==============================] - 0s 7ms/step - loss: 0.2212 - accuracy: 0.9032 - val_loss: 0.4351 - val_accuracy: 0.8963
Epoch 91/100
20/20 [==============================] - 0s 10ms/step - loss: 0.2080 - accuracy: 0.9206 - val_loss: 0.4121 - val_accuracy: 0.9000
Epoch 92/100
20/20 [==============================] - 0s 8ms/step - loss: 0.1964 - accuracy: 0.9159 - val_loss: 0.4236 - val_accuracy: 0.9111
Epoch 93/100
20/20 [==============================] - 0s 8ms/step - loss: 0.2191 - accuracy: 0.9190 - val_loss: 0.4497 - val_accuracy: 0.9111
Epoch 94/100
20/20 [==============================] - 0s 9ms/step - loss: 0.1915 - accuracy: 0.9302 - val_loss: 0.4726 - val_accuracy: 0.9074
Epoch 95/100
20/20 [==============================] - 0s 7ms/step - loss: 0.2401 - accuracy: 0.9127 - val_loss: 0.4470 - val_accuracy: 0.9000
Epoch 96/100
20/20 [==============================] - 0s 8ms/step - loss: 0.2035 - accuracy: 0.9238 - val_loss: 0.4640 - val_accuracy: 0.9074
Epoch 97/100
20/20 [==============================] - 0s 9ms/step - loss: 0.2249 - accuracy: 0.9016 - val_loss: 0.4682 - val_accuracy: 0.8889
Epoch 98/100
20/20 [==============================] - 0s 10ms/step - loss: 0.2461 - accuracy: 0.9111 - val_loss: 0.4774 - val_accuracy: 0.9074
Epoch 99/100
20/20 [==============================] - 0s 8ms/step - loss: 0.2390 - accuracy: 0.9079 - val_loss: 0.4781 - val_accuracy: 0.8926
Epoch 100/100
20/20 [==============================] - 0s 9ms/step - loss: 0.1935 - accuracy: 0.9238 - val_loss: 0.4689 - val_accuracy: 0.9037
```

```
history_df = pd.DataFrame(history.history)
history_df[['accuracy', 'val_accuracy']].plot()
```

```
<Axes: >
```

```
history_df = pd.DataFrame(history.history)
history_df[['loss', 'val_loss']].plot()
```

```
<Axes: >
```




```
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
```

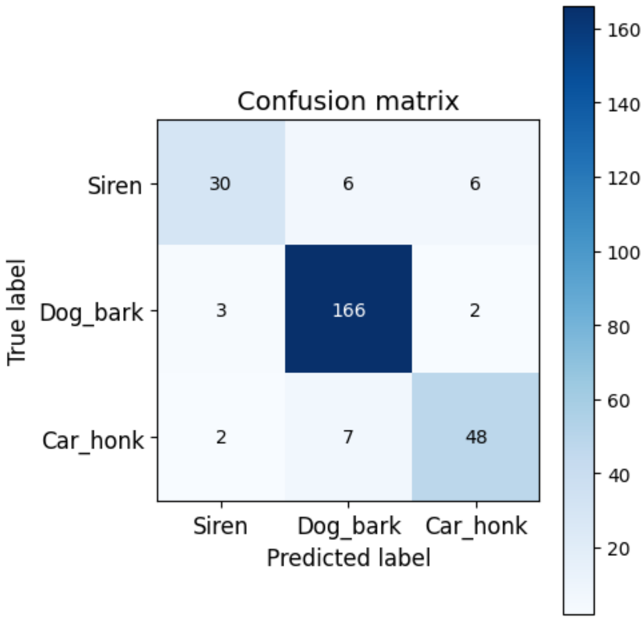
```
9/9 [==============================] - 0s 2ms/step - loss: 0.4689 - accuracy: 0.9037
Test accuracy: 0.9037036895751953
```

```
# Generate predictions on the test set
y_pred = model.predict(X_test)

# Convert predictions to class labels
y_pred_classes = np.argmax(y_pred, axis=1)

# Convert predictions to class labels
y_test_classes = np.argmax(y_test, axis=1)

# Print the classification report
print(classification_report(y_test_classes, y_pred_classes, target_names=['Siren', 'Dog_bark', 'Car_honk']))
```

```
9/9 [==============================] - 0s 2ms/step
              precision    recall  f1-score   support

       Siren       0.86      0.71      0.78        42
    Dog_bark       0.93      0.97      0.95       171
    Car_honk       0.86      0.84      0.85        57

    accuracy                           0.90       270
   macro avg       0.88      0.84      0.86       270
weighted avg       0.90      0.90      0.90       270
```

```
class_labels = ['Siren', 'Dog_bark', 'Car_honk']
# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test_classes, y_pred_classes)

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))
im = ax.imshow(conf_matrix, cmap="Blues")

# Add colorbar
cbar = ax.figure.colorbar(im, ax=ax)

# Add labels to the x-axis and y-axis
ax.set_xticks(np.arange(len(class_labels)))
ax.set_yticks(np.arange(len(class_labels)))
ax.set_xticklabels(class_labels, fontsize=12)
ax.set_yticklabels(class_labels, fontsize=12)
```

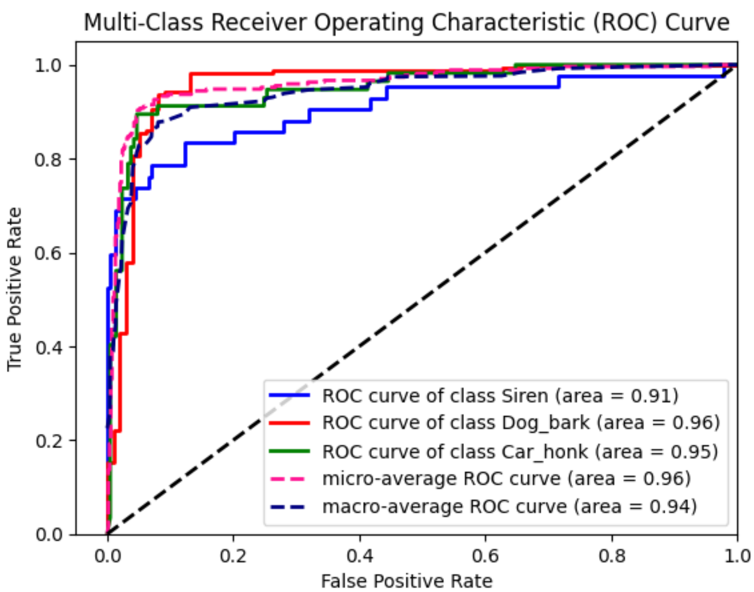
```
# Add annotations
thresh = conf_matrix.max() / 2.0
for i in range(len(class_labels)):
    for j in range(len(class_labels)):
        ax.text(j, i, format(conf_matrix[i, j], "d"),
                ha="center", va="center",
                color="white" if conf_matrix[i, j] > thresh else "black")

# Add title and axis labels
ax.set_title("Confusion matrix", fontsize=14)
ax.set_xlabel("Predicted label", fontsize=12)
ax.set_ylabel("True label", fontsize=12)

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




```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize

# Binarize the true labels
y_test_binarized = label_binarize(y_test, classes=[0, 1, 2])

classes = class_labels

# Compute the predicted class probabilities for each sample
y_pred_prob = model.predict(X_test)

# Compute the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(classes)):
    fpr[i], tpr[i], _ = roc_curve(y_test_binarized[:, i], y_pred_prob[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and AUC
fpr_micro, tpr_micro, _ = roc_curve(y_test_binarized.ravel(), y_pred_prob.ravel())
roc_auc_micro = auc(fpr_micro, tpr_micro)

# Compute macro-average ROC curve and AUC
fpr_macro = np.unique(np.concatenate([fpr[i] for i in range(len(classes))]))
tpr_macro = np.zeros_like(fpr_macro)
for i in range(len(classes)):
    tpr_macro += np.interp(fpr_macro, fpr[i], tpr[i])
tpr_macro /= len(classes)
roc_auc_macro = auc(fpr_macro, tpr_macro)

# Plot the ROC curves for each class
plt.figure()
```

```
colors = ['blue', 'red', 'green']
for i, color in zip(range(len(classes)), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=2,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(classes[i], roc_auc[i]))

# Plot the micro-average ROC curve
plt.plot(fpr_micro, tpr_micro, color='deeppink', lw=2, linestyle='--',
         label='micro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc_micro))

# Plot the macro-average ROC curve
plt.plot(fpr_macro, tpr_macro, color='navy', lw=2, linestyle='--',
         label='macro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc_macro))

# Add labels and legend
plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Multi-Class Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

```
9/9 [==============================] - 0s 4ms/step
```




```
model.save('model.h5')
```

Prediction Function

```
def predict_class(audio_path):
    n_mfcc = 20

    # Load audio file
    audio = AudioSegment.from_file(audio_path)
    audio_data = np.array(audio.get_array_of_samples())
    sample_rate = audio.frame_rate

    # Split audio into chunks of silence
    audio_chunks = split_on_silence(audio, min_silence_len=500, silence_thresh=-50)

    # Extract MFCCs from each chunk of audio
    mfccs_list = []
    nfft = 2048 # Set larger FFT size
    for chunk in audio_chunks:
        chunk_data = np.array(chunk.get_array_of_samples())
```

```
        mfccs = mfcc(chunk_data, samplerate=sample_rate, numcep=n_mfcc, nfft=nfft) # Set larger nfft value
        mfccs = np.mean(mfccs, axis=0)
        mfccs_list.append(mfccs)

    # Convert MFCCs to a numpy array
    mfccs_array = np.array(mfccs_list)

    # Reshape MFCCs to have 1 channel
    mfccs_array = mfccs_array.reshape(mfccs_array.shape[0], mfccs_array.shape[1], 1)

    # Load trained model
    model = load_model('model.h5')

    # Get predicted probabilities for each class
    probs = model.predict(mfccs_array)

    # Convert probabilities to predicted class labels
    predicted_classes = np.argmax(probs, axis=1)

    # Convert predicted class labels to class names
    class_names = ['Siren', 'Dog_bark', 'Car_honk']
    predicted_class_names = [class_names[i] for i in predicted_classes]

    return predicted_class_names
```

```
predict_class('/content/drive/MyDrive/audio/Car_honk/235506__ceberation__car-horn.wav')
```

```
1/1 [==============================] - 0s 65ms/step
['Car_honk']
```

```
predict_class('/content/drive/MyDrive/audio/Dog_bark/100032__nfrae__rose_bark.wav')
```

```
1/1 [==============================] - 0s 72ms/step
['Dog_bark']
```

```
predict_class('/content/drive/MyDrive/audio/Siren/108192__paubg_pou__ambulancia.wav')
```

```
1/1 [==============================] - 0s 72ms/step
['Siren']
```

✓ 2s completed at 10:30 PM