



**The University of Azad Jammu and Kashmir Muzaffarabad**

**Department of Software Engineering**

***Semester Project***

|  |  |
| --- | --- |
| *Course:* | *Computer Vision* |
| *Course Code:* | ***SE-4103*** |
| *Submitted To:* | ***Engr. Ahmad Khawaja*** |
| *Submitted By:* | ***Jawad Sheikh, Murtaza Haider, Baqir Ali*** |
| *Roll No:* | ***20-SE-02, 20-SE-16, 20-SE-38*** |
| *Session:* | ***2020-2024*** |
| *Semester:* | ***Sem-07*** |

|  |
| --- |
| Table of Content  [Installing Required Libraries 3](#_Toc162178251)  [Importing Required Libraries 3](#_Toc162178252)  [File Extraction 4](#_Toc162178253)  [Loading and Dividing Dataset 5](#_Toc162178254)  [Features Extraction 7](#_Toc162178255)  [Model Building 9](#_Toc162178256)  [Model loading, Compiling and Training 11](#_Toc162178257)  [Model Evaluation and Visualization 13](#_Toc162178258) |

**Explanation of code**

# Installing Required Libraries

!pip uninstall -y librosa resampy

This command uninstalls the packages "**librosa**" and "**resampy**" from the current Python environment. The **–y** flag is used to automatically confirm uninstallation without prompting for user confirmation.

!pip install librosa resampy

This command installs the packages "**librosa**" and "**resampy**" into the current Python environment, ensuring they are present and up-to-date.

pip install py7zr

The command installs the Python package called "**py7zr**" into the current Python environment. This package provides functionality for working with 7z archives in Python.

# Importing Required Libraries

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.layers import Dropout

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import f1\_score, roc\_curve, auc, confusion\_matrix, classification\_report, precision\_recall\_curve, average\_precision\_score

from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt

import py7zr

import librosa

import os

This code imports several Python libraries and modules:

* **`numpy` (aliased as `np`):** A library for numerical computing in Python.
* **`tensorflow` (aliased as `tf`):** An open-source machine learning framework for building and training neural networks.
* **`layers` and `models` from `tensorflow.keras`:** Sub-modules within TensorFlow's high-level API for building neural networks.
* **`Dropout` from `tensorflow.keras.layers`:** A layer for implementing dropout regularization in neural networks.
* **`train\_test\_split` from `sklearn.model\_selection`:** A function for splitting datasets into train and test sets.
* **Various evaluation metrics and tools from `sklearn.metrics`:** Functions for evaluating model performance, including f1\_score, roc\_curve, auc, confusion\_matrix, classification\_report, precision\_recall\_curve, and average\_precision\_score.
* **`LabelEncoder` from `sklearn.preprocessing`:** A class for encoding categorical labels as numerical values.
* **`matplotlib.pyplot` (aliased as `plt`):** A plotting library for creating visualizations in Python.
* **`py7zr`:** A Python library for working with 7z archives.
* **`librosa`:** A Python library for audio and music analysis.
* **`os`:** A module for interacting with the operating system, used for file operations and directory navigation.

# File Extraction

# Path to your .7z file

file\_path = '/content/drive/MyDrive/cv\_project/samples.7z'

# Directory to extract the contents of the .7z file

extracted\_dir = '/content/drive/MyDrive/cv\_project/'

# Create the directory if it doesn't exist

os.makedirs(extracted\_dir, exist\_ok=True)

# Extract the contents of the .7z file

with py7zr.SevenZipFile(file\_path, mode='r') as z:

    z.extractall(path=extracted\_dir)

print("Extraction completed.")

This code extracts the contents of a .7z file located at the specified **`file\_path`** to the directory specified by **`extracted\_dir`.** Here's a breakdown of the code:

* **`file\_path`:** Specifies the path to the .7z file that needs to be extracted.
* **`extracted\_dir`:** Specifies the directory where the contents of the .7z file will be extracted.
* **`os.makedirs(extracted\_dir, exist\_ok=True)`:** Creates the directory specified by `extracted\_dir` if it doesn't already exist. The `exist\_ok=True` parameter ensures that the function does not raise an error if the directory already exists.
* **`with py7zr.SevenZipFile(file\_path, mode='r') as z:`:** Opens the .7z file specified by `file\_path` in read-only mode using the `py7zr` library. The `with` statement ensures that the file is properly closed after extraction.
* **`z.extractall(path=extracted\_dir)`:** Extracts all contents of the .7z file to the directory specified by `extracted\_dir`.
* Finally, it prints a message indicating that the extraction process is completed.

This code effectively extracts the contents of the .7z file to the specified directory.

# Loading and Dividing Dataset

def load\_data(test\_size=0.2, chunk\_duration=1):

    x, y = [], []

    for file in os.listdir('/content/drive/MyDrive/cv\_project/samples'):

        # Load audio file

        audio, sample\_rate = librosa.load('/content/drive/MyDrive/cv\_project/samples/' + file, res\_type='kaiser\_fast')

        # Calculate number of chunks

        num\_chunks = int(np.ceil(len(audio) / (sample\_rate \* chunk\_duration)))

        # Extract features from each chunk

        for i in range(num\_chunks):

            start = int(i \* sample\_rate \* chunk\_duration)

            end = min(len(audio), int((i + 1) \* sample\_rate \* chunk\_duration))

            chunk\_audio = audio[start:end]

            # Extract features from audio chunk

            feature = extract\_features\_from\_audio(chunk\_audio, sample\_rate)

            x.append(feature)

            # Extract class label from the file name

            class\_label = file.split('(')[0]  # Assuming the class label is before the first '-'

            y.append(class\_label)

    # Encode the labels

    encoder = LabelEncoder()

    y = encoder.fit\_transform(y)

    return train\_test\_split(np.array(x), y, test\_size=test\_size, random\_state=42)

This function **`load\_data`** is designed to load audio data from files stored in a specified directory, extract features from each audio chunk, and prepare the data for training a machine learning model. Here's a breakdown of the function:

* **`test\_size=0.2, chunk\_duration=1`:** These are default parameters for specifying the size of the test set (default is 20%) and the duration of each audio chunk in seconds (default is 1 second).
* **`x, y = [], []`:** Initialize empty lists to store features (x) and labels (y).
* **`for file in os.listdir('/content/drive/MyDrive/cv\_project/samples'):`:** Iterates through all files in the specified directory containing audio samples.
* **`audio, sample\_rate = librosa.load('/content/drive/MyDrive/cv\_project/samples/' + file, res\_type='kaiser\_fast')`:** Loads the audio file using librosa library, specifying a faster resampling method ('kaiser\_fast').
* **`num\_chunks = int(np.ceil(len(audio) / (sample\_rate \* chunk\_duration)))`:** Calculates the number of chunks required to cover the entire audio, based on the specified chunk duration.
* Inside the nested loop:
  + **`start = int(i \* sample\_rate \* chunk\_duration)`:** Calculates the starting index of the current audio chunk.
  + **`end = min(len(audio), int((i + 1) \* sample\_rate \* chunk\_duration))`:** Calculates the ending index of the current audio chunk, ensuring it doesn't exceed the length of the audio.
  + **`chunk\_audio = audio[start:end]`:** Extracts the current audio chunk.
  + **`feature = extract\_features\_from\_audio(chunk\_audio, sample\_rate)`:** Calls a function `extract\_features\_from\_audio` to extract features from the audio chunk.
  + **`x.append(feature)`:** Appends the extracted feature to the list of features.
  + **`class\_label = file.split('(')[0]`:** Extracts the class label from the file name assuming a certain naming convention (class label appears before the first '(' character).
  + **`y.append(class\_label)`:** Appends the class label to the list of labels.
* **`encoder = LabelEncoder()`:** Initializes a label encoder object.
* **`y = encoder.fit\_transform(y)`:** Encodes the class labels into numerical format.
* Finally, it returns the train-test split of the features (x) and labels (y) using `train\_test\_split` from scikit-learn, ensuring reproducibility by setting `random\_state=42`.

# Features Extraction

def extract\_features\_from\_audio(audio, sample\_rate, mfcc=True, chroma=True, mel=True, zero\_crossing\_rate=True, spectral\_bandwidth=True, statistic='square\_root\_sum'):

    result = []

    if mfcc:

        mfccs = librosa.feature.mfcc(y=audio, sr=sample\_rate, n\_mfcc=40, n\_fft=1024)

        if statistic == 'square\_root\_sum':

            mfccs = np.sqrt(np.sum(np.square(mfccs), axis=1))

        # Add other cases for different statistics if needed

        result.append(mfccs)

    if chroma:

        stft = np.abs(librosa.stft(audio, n\_fft=1024))

        chroma = librosa.feature.chroma\_stft(S=stft, sr=sample\_rate)

        if statistic == 'square\_root\_sum':

            chroma = np.sqrt(np.sum(np.square(chroma), axis=1))

        # Add other cases for different statistics if needed

        result.append(chroma)

    if mel:

        mel = librosa.feature.melspectrogram(y=audio, sr=sample\_rate, n\_fft=1024)

        if statistic == 'square\_root\_sum':

            mel = np.sqrt(np.sum(np.square(mel), axis=1))

        # Add other cases for different statistics if needed

        result.append(mel)

    if zero\_crossing\_rate:

        zcr = librosa.feature.zero\_crossing\_rate(audio)

        if statistic == 'square\_root\_sum':

            zcr = np.sqrt(np.sum(np.square(zcr), axis=1))

        # Add other cases for different statistics if needed

        result.append(zcr)

    if spectral\_bandwidth:

        spec\_bw = librosa.feature.spectral\_bandwidth(y=audio, sr=sample\_rate)

        if statistic == 'square\_root\_sum':

            spec\_bw = np.sqrt(np.sum(np.square(spec\_bw), axis=1))

        # Add other cases for different statistics if needed

        result.append(spec\_bw)

    return np.hstack(result)

This function, **`extract\_features\_from\_audio`,** is responsible for extracting various features from an audio signal. Here's how it works:

* Inputs
  + **`audio`:** The audio signal.
  + **`sample\_rate`:** The sampling rate of the audio signal.
  + **`mfcc`, `chroma`, `mel`, `zero\_crossing\_rate`, `spectral\_bandwidth`:** Boolean flags indicating which features to extract. These features include Mel-Frequency Cepstral Coefficients (MFCC), chroma features, mel spectrogram, zero-crossing rate, and spectral bandwidth.
  + **`statistic`:** Specifies the type of statistic to apply to the extracted features. The default is 'square\_root\_sum'.
* Output
  + Returns a concatenated array of the extracted features.
* Feature Extraction
  + For each feature type enabled (`mfcc`, `chroma`, etc.), the function calculates the corresponding feature using functions from the `librosa` library.
  + It applies the specified statistic (`square\_root\_sum` by default) to the feature if needed.
  + The resulting features are appended to a list (`result`).
  + Finally, the features are horizontally stacked (concatenated) using `np.hstack` and returned as a single array.

This function allows for flexibility in feature extraction by enabling or disabling different types of features and choosing different statistical summaries of those features.

# Model Building

def create\_model\_with\_dropout(input\_shape, num\_classes, dropout\_rate=0.5):

    model = models.Sequential()

    model.add(layers.Conv1D(64, 3, activation='relu', input\_shape=input\_shape))

    model.add(layers.MaxPooling1D(2))

    model.add(layers.Conv1D(128, 3, activation='relu'))

    model.add(layers.MaxPooling1D(2))

    model.add(layers.Conv1D(256, 3, activation='relu'))

    model.add(layers.MaxPooling1D(2))

    model.add(layers.Conv1D(256, 3, activation='relu'))

    model.add(layers.MaxPooling1D(2))

    model.add(layers.Flatten())

    model.add(layers.Dense(512, activation='relu'))

    model.add(Dropout(dropout\_rate))  # Adding dropout layer

    model.add(layers.Dense(num\_classes, activation='softmax'))

    return model

This **`create\_model\_with\_dropout`** function constructs a convolutional neural network (CNN) model with dropout regularization. Here's a breakdown of the model architecture:

* Input Shape
  + **`input\_shape`:** Specifies the shape of the input data. It is expected to be a tuple representing the dimensions of the input data. In this case, it is assumed to be `(number\_of\_features, 1)` where `number\_of\_features` represents the number of features in each input sample.
* Layers
  + `Conv1D` Layer:
    - 64 filters with a kernel size of 3 and ReLU activation function.
    - Input shape is determined by `input\_shape`.
  + `MaxPooling1D` Layer:
    - Pooling layer with pool size 2.
  + Another set of `Conv1D` and `MaxPooling1D` layers with increased filter sizes (128, 256, 256).
  + `Flatten` Layer:
    - Flattens the output of the previous layer into a one-dimensional vector.
  + `Dense` Layer:
    - Fully connected layer with 512 units and ReLU activation function.
  + `Dropout` Layer:
    - Applies dropout regularization with the specified `dropout\_rate`.
  + Output `Dense` Layer:
  + Output layer with `num\_classes` units and softmax activation function, suitable for multi-class classification.
* Parameters
  + **`input\_shape`:** Shape of the input data.
  + **`num\_classes`:** Number of classes in the classification task.
  + **`dropout\_rate`:** Dropout rate, controlling the proportion of input units to drop during training.
* Return
  + Returns the constructed model.

This architecture is commonly used for processing one-dimensional sequential data such as time-series or audio signals. The inclusion of dropout layers helps in preventing overfitting by randomly dropping a fraction of input units during training.

# Model loading, Compiling and Training

    # Load and split the dataset

X\_train, X\_test, y\_train, y\_test = load\_data(test\_size=0.2)

# Update input\_shape to have 2 dimensions (samples, time\_steps)

input\_shape = (X\_train.shape[1], 1)

num\_classes=4

# Create the model with dropout

model\_with\_dropout = create\_model\_with\_dropout(input\_shape=(X\_train.shape[1], 1), num\_classes=num\_classes)

# Compile the model

model\_with\_dropout.compile(optimizer='adam',

                           loss='sparse\_categorical\_crossentropy',

                           metrics=['accuracy'])

# Display model summary

model\_with\_dropout.summary()

# Train the model

model\_with\_dropout.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_data=(X\_test, y\_test))

# Evaluate the model

test\_loss, test\_acc = model\_with\_dropout.evaluate(X\_test, y\_test)

print('Test accuracy:', test\_acc)

This code snippet loads and splits the dataset, creates a CNN model with dropout, compiles the model, trains it, and evaluates its performance. Here's a breakdown:

1. Loading and Splitting the Dataset

* **`X\_train, X\_test, y\_train, y\_test = load\_data(test\_size=0.2)`:** Loads and splits the dataset into training and testing sets. The `test\_size` parameter is set to 20% to allocate 20% of the data for testing.

2. Updating Input Shape

* **`input\_shape = (X\_train.shape[1], 1)`:** Updates the input shape to have two dimensions: `(number\_of\_features, 1)`. It assumes that the input data has a single time step.

3. Creating the Model

* **`model\_with\_dropout = create\_model\_with\_dropout(input\_shape=(X\_train.shape[1], 1), num\_classes=num\_classes)`:** Creates a CNN model with dropout regularization using the `create\_model\_with\_dropout` function. The input shape and number of classes are specified.

4. Compiling the Model

* + **`model\_with\_dropout.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])`:** Compiles the model with the Adam optimizer, sparse categorical cross-entropy loss function, and accuracy metric.

5. Displaying Model Summary

* + **`model\_with\_dropout.summary()`:** Prints the summary of the model architecture.

6. Training the Model

* + - **`model\_with\_dropout.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_data=(X\_test, y\_test))`:** Trains the model for 50 epochs with a batch size of 32 using the training data and validates it using the test data.

7. Evaluating the Model

* + - * **`test\_loss, test\_acc = model\_with\_dropout.evaluate(X\_test, y\_test**)`: Evaluates the trained model on the test data, obtaining the test loss and accuracy.
      * **`print('Test accuracy:', test\_acc)`:** Prints the test accuracy obtained from the evaluation.

This code provides a complete workflow for training and evaluating the CNN model with dropout regularization on the provided dataset.

# Model Evaluation and Visualization

# Calculate F1 score

f1 = f1\_score(y\_test, y\_pred\_labels, average='weighted')

print('F1 Score:', f1)

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_labels)

print('Confusion Matrix:')

print(conf\_matrix)

# Display classification report

print('Classification Report:')

print(classification\_report(y\_test, y\_pred\_labels))

Calculate F1 Score:

* **f1 = f1\_score(y\_test, y\_pred\_labels, average='weighted'): Calculates** the F1 score, which is a harmonic mean of precision and recall, for the test data. The average='weighted' parameter specifies the averaging method for multi-class classification.

Generate Confusion Matrix:

* **conf\_matrix = confusion\_matrix(y\_test, y\_pred\_labels):** Generates a confusion matrix to visualize the performance of the classifier on each class.

Display Classification Report:

* **print('Classification Report:'):** Prints a classification report containing precision, recall, F1-score, and support for each class.
* **print(classification\_report(y\_test, y\_pred\_labels)):** Displays the classification report.

**ROC curve**

# Plot ROC curve for each class

plt.figure(figsize=(8, 6))

for i in range(num\_classes):

    if np.sum(y\_test == i) > 0:  # Check if the class has positive samples

        fpr, tpr, \_ = roc\_curve(y\_test == i, y\_pred\_proba[:, i])

        roc\_auc = auc(fpr, tpr)

        plt.plot(fpr, tpr, label=f'Class {i} (AUC = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], linestyle='--', color='gray')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve')

plt.legend()

plt.grid(True)

plt.show()

This code snippet plots the Receiver Operating Characteristic (ROC) curve for each class in a multi-class classification problem. Here's a breakdown of the code:

1. Plotting ROC Curves

* + - * The code iterates over each class (`i`) in the range of `num\_classes`.
      * For each class, it checks if there are positive samples (`np.sum(y\_test == i) > 0`). If there are no positive samples for a class, ROC curve cannot be plotted for that class.
      * If positive samples exist, it calculates the False Positive Rate (FPR) and True Positive Rate (TPR) using the `roc\_curve` function from scikit-learn.
      * It also calculates the Area Under the Curve (AUC) for each class using the `auc` function from scikit-learn.
      * It then plots the ROC curve for each class, labeling it with the class number and its corresponding AUC value.

2. Plot Customization

* + - * It plots a dashed diagonal line representing random guessing.
      * Labels axes and provides a title for the plot.
      * Adds a legend to the plot showing the class number and its AUC value.
      * Displays the grid.

3. Displaying the Plot

* Finally, it displays the ROC curve plot using `plt.show()`.

This plot helps visualize the performance of the classifier for each class in terms of its ability to distinguish between positive and negative samples. The AUC value provides a quantitative measure of the classifier's performance, with higher values indicating better performance.

**Precision-Recall Curve**

# Precision-Recall Curve for each class

plt.figure(figsize=(8, 6))

for i in range(num\_classes):

    precision, recall, \_ = precision\_recall\_curve(y\_test == i, y\_pred\_proba[:, i])

    average\_precision = average\_precision\_score(y\_test == i, y\_pred\_proba[:, i])

    plt.plot(recall, precision, label=f'Class {i} (AP = {average\_precision:.2f})')

plt.xlabel('Recall')

plt.ylabel('Precision')

plt.title('Precision-Recall Curve for each class')

plt.legend()

plt.grid(True)

plt.show()

This code snippet plots the Precision-Recall curve for each class in a multi-class classification problem. Here's how it works:

1. Plotting Precision-Recall Curves

* + - * The code iterates over each class (`i`) in the range of `num\_classes`.
      * For each class, it calculates the Precision and Recall using the `precision\_recall\_curve` function from scikit-learn.
      * It also calculates the Average Precision (AP) for each class using the `average\_precision\_score` function from scikit-learn.
      * It then plots the Precision-Recall curve for each class, labeling it with the class number and its corresponding AP value.

2. Plot Customization

* + - * Labels axes and provides a title for the plot.
      * Adds a legend to the plot showing the class number and its AP value.
      * Displays the grid.

3. Displaying the Plot

* Finally, it displays the Precision-Recall curve plot using `plt.show()`.

This plot helps evaluate the classifier's performance for each class in terms of both precision and recall. The AP value provides a summary measure of the Precision-Recall curve, with higher values indicating better performance.