Multimodal System for Suspicious Emotions Config

System Requirements:

RAM: 16GB
OS: Windows 13 or Ubuntu
Environment: Google Colab

Steps to Run:

1. Log Into the Github
2. Click on the link of the streamlit
3. Upload the files and predict

Text Analysis

The code begins by installing essential libraries:

```
!pip install nltk scikit-learn xgboost
```

We import a suite of necessary libraries:

```
import pandas as pd
import numpy as np
import re
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

Next, we download essential NLTK datasets for handling stopwords and tokenizing text:

```
nltk.download('stopwords')
nltk.download('punkt_tab')
```

We load the WASSA 2017 dataset from a specified URL into a Pandas DataFrame and display an overview of the dataset:

```
wassa_url='https://archive.org/download/misc-dataset/wassa-2017.csv'
df=pd.read_csv(wassa_url)
df.info()
df
```

Irrelevant columns, such as 'rid' and 'tid', are dropped, and the structure and initial rows of the dataset are examined:

```
df.drop(['rid', 'tid'], axis=1, inplace=True)
df.shape
df.head()
```

Text preprocessing functions are defined to clean tweets by removing mentions, special characters, and links, and to tokenize the text, convert it to lowercase, and remove stopwords:

```
def clean_tweet(tweet):
    tweet = re.sub(r"(@[A-Za-z0-9_]+)|([^0-9A-Za-z \t])|(\w+:\/\/\S+)", "
", str(tweet))
    tweet = re.sub(r"\s+", " ", tweet).strip()
    return tweet

def nltk_preprocess(text):
    stop_words = set(stopwords.words('english'))
    words = word_tokenize(text.lower())
    filtered_words = [word for word in words if word.isalnum() and word not
in stop_words]
    return " ".join(filtered_words)
```

These functions are applied to each tweet in the dataset, resulting in cleaned and tokenized text:

```
df['cleaned_tweet'] = df['text'].apply(clean_tweet)
df['processed_text'] = df['cleaned_tweet'].apply(nltk_preprocess)
```

Text features are then extracted using the CountVectorizer, which transforms the cleaned text into a matrix of token counts. The intensity column is reshaped to combine with the text features:

```
vectorizer = CountVectorizer()
X_text = vectorizer.fit_transform(df['processed_text'])
X_intensity = np.array(df['intensity']).reshape(-1, 1)
```

These features are merged into a single feature matrix using sparse matrix stacking:

```
from scipy.sparse import hstack
X = hstack((X_text, X_intensity))
```

Emotion labels are converted from strings to numeric values to prepare for model training:

```
label_mapping = {'anger': 0, 'fear': 1, 'joy': 2, 'sadness': 3}
inverse_label_mapping = {v: k for k, v in label_mapping.items()}
df['numeric_emotion'] = df['emotion'].map(label_mapping)
y = df['numeric_emotion']
```

The distribution of different emotions in the dataset is visualized using a count plot:

```
plt.figure(figsize=(8, 6))
sns.countplot(x='emotion', data=df, palette='Set2')
plt.title('Distribution of Emotions')
plt.xlabel('Emotion')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```

The data is split into training and testing sets, reserving 20% for testing:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Three machine learning models are defined and trained: Logistic Regression, Random Forest, and XGBoost. Each model is fitted to the training data, and predictions are made on the test set. The performance of each model is evaluated using accuracy, classification reports, and confusion matrices, which are visualized for clarity:

```
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(n_estimators=100),
    "XGBoost": XGBClassifier(use_label_encoder=False,
eval_metric='mlogloss')
}

for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_pred_labels = [inverse_label_mapping[pred] for pred in y_pred]
    y_test_labels = [inverse_label_mapping[true] for true in y_test]

    print(f"\n--- {name} ---")
    print("Accuracy:", accuracy_score(y_test_labels, y_pred_labels))
    print("Classification Report:\n", classification_report(y_test_labels,
y_pred_labels))

    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=list(label_mapping.keys()))
    disp.plot(cmap=plt.cm.Blues)
    plt.title(f'Confusion Matrix - {name}')
    plt.show()
```

Image Analysis

This section describes the methodology and code implementation for analyzing facial expressions using various deep learning models, namely DenseNet121, EfficientNetB7, MobileNet, VGG19, and ResNet50. The analysis is performed on the FER-2013 dataset, focusing on detecting emotions from facial images.

We begin by importing the necessary libraries for data manipulation, image processing, machine learning, and visualization:

```
import math
import numpy as np
import pandas as pd
import cv2
import seaborn as sns
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
import tensorflow as tf
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Flatten, Dense, Conv2D,
GlobalAveragePooling2D
from tensorflow.keras.layers import Dropout, BatchNormalization, Activation
from tensorflow.keras.callbacks import Callback, EarlyStopping,
ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.utils import to_categorical
```

The FER-2013 dataset is loaded into a pandas DataFrame and the initial structure is examined:

```
df = pd.read_csv('/content/fer2013.csv')
print(df.shape)
df.head()
df.emotion.unique()
```

We define a mapping of emotion labels to textual descriptions and visualize the distribution of emotion classes within the dataset:

```
emotion_label_to_text = {0: 'anger', 1: 'disgust', 2: 'fear', 3:
'happiness', 4: 'sadness', 5: 'surprise', 6: 'neutral'}
df.emotion.value_counts()
```

A sample of images from each emotion class is visualized to provide an overview of the dataset:

```
fig = plt.figure(1, (14, 14))
k = 0
for label in sorted(df.emotion.unique()):
    for j in range(7):
        px = df[df.emotion == label].pixels.iloc[k]
        px = np.array(px.split(' ')).reshape(48, 48).astype('float32')
        k += 1
        ax = plt.subplot(7, 7, k)
        ax.imshow(px, cmap='gray')
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(emotion_label_to_text[label])
        plt.tight_layout()
```

The pixel values are converted into a NumPy array and the images are transformed to RGB format:

```
img_array = df.pixels.apply(lambda x: np.array(x.split(' ')).reshape(48,
48).astype('float32'))
img_array = np.stack(img_array, axis=0)
img_features = []

for i in range(len(img_array)):
    temp = cv2.cvtColor(img_array[i], cv2.COLOR_GRAY2RGB)
    img_features.append(temp)

img_features = np.array(img_features)
print(img_features.shape)
plt.imshow(img_features[0].astype(np.uint8))
```

We encode the emotion labels into categorical format:

```
le = LabelEncoder()
img_labels = le.fit_transform(df.emotion)
img_labels = to_categorical(img_labels)
img_labels.shape
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print(le_name_mapping)
```

The dataset is split into training and validation sets:

```
X_train, X_valid, y_train, y_valid = train_test_split(img_features,
img_labels, shuffle=True, stratify=img_labels, test_size=0.1,
random_state=42)
X_train.shape, X_valid.shape, y_train.shape, y_valid.shape
```

The image data is normalized to enhance the performance of neural networks:

```
img_width = X_train.shape[1]
img_height = X_train.shape[2]
img_depth = X_train.shape[3]
num_classes = y_train.shape[1]

X_train = X_train / 255.
X_valid = X_valid / 255.
```

In the following section, we demonstrate the process using the DenseNet121 model. The same structure is applied to other models such as EfficientNetB7, MobileNet, VGG19, and ResNet50, with differences only in the model architecture initialization.

We initialize the DenseNet121 model pre-trained on ImageNet:

```
densenet121 = tf.keras.applications.DenseNet121(weights='imagenet',
include_top=False, input_shape=(48, 48, 3))
densenet121.summary()
```

We define a function to build the top model:

```
def build_model(bottom_model, classes):
    model = bottom_model.layers[-2].output
    model = GlobalAveragePooling2D()(model)
    model = Dense(classes, activation='softmax', name='out_layer')(model)
    return model

head = build_model(densenet121, num_classes)
densenet121_model = Model(inputs=densenet121.input, outputs=head)
print(densenet121_model.summary())
```

Callbacks for early stopping and learning rate reduction are set up:

```
early_stopping = EarlyStopping(monitor='val_accuracy', min_delta=0.00005,
patience=11, verbose=1, restore_best_weights=True)
lr_scheduler = ReduceLROnPlateau(monitor='val_accuracy', factor=0.5,
patience=7, min_lr=1e-7, verbose=1)
callbacks = [early_stopping, lr_scheduler]
```

We configure the image data generator for augmentation:

```
train_datagen = ImageDataGenerator(rotation_range=15,
width_shift_range=0.15, height_shift_range=0.15, shear_range=0.15,
zoom_range=0.15, horizontal_flip=True)
train_datagen.fit(X_train)
```

The model is compiled and trained:

```
batch_size = 32
epochs = 25
optims = [optimizers.Adam(learning_rate=0.0001, beta_1=0.9, beta_2=0.999)]
densenet121_model.compile(loss='categorical_crossentropy',
optimizer=optims[0], metrics=['accuracy'])

history = densenet121_model.fit(train_datagen.flow(X_train, y_train,
batch_size=batch_size), validation_data=(X_valid, y_valid),
steps_per_epoch=len(X_train) // batch_size, epochs=epochs,
callbacks=callbacks)
```

We visualize the training and validation accuracy and loss:

```
sns.set()
fig = plt.figure(0, (12, 4))

ax = plt.subplot(1, 2, 1)
sns.lineplot(x=history.epoch, y=history.history['accuracy'], label='train')
sns.lineplot(x=history.epoch, y=history.history['val_accuracy'],
label='valid')
plt.title('Accuracy')
plt.tight_layout()

ax = plt.subplot(1, 2, 2)
sns.lineplot(x=history.epoch, y=history.history['loss'], label='train')
sns.lineplot(x=history.epoch, y=history.history['val_loss'], label='valid')
plt.title('Loss')
plt.tight_layout()
plt.show()
```

We plot violin plots for a more comprehensive view of accuracy and loss distribution:

```
df_accu = pd.DataFrame({'train': history.history['accuracy'], 'valid':
history.history['val_accuracy']})
df_loss = pd.DataFrame({'train': history.history['loss'], 'valid':
history.history['val_loss']})

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 4))

sns.violinplot(data=pd.melt(df_accu), x="variable", y="value", ax=ax1)
ax1.set_title('Accuracy')
ax1.set_xlabel('')
ax1.set_ylabel('Accuracy')
```

```
sns.violinplot(data=pd.melt(df_loss), x="variable", y="value", ax=ax2)
ax2.set_title('Loss')
ax2.set_xlabel('')
ax2.set_ylabel('Loss')

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

Predictions on the validation set are evaluated and visualized with a confusion matrix:

```
yhat_valid = np.argmax(densenet121_model.predict(X_valid), axis=1)
print(f'total wrong validation predictions: {np.sum(np.argmax(y_valid,
axis=1) != yhat_valid)}\n\n')
print(classification_report(np.argmax(y_valid, axis=1), yhat_valid))

cf_matrix = confusion_matrix(np.argmax(y_valid, axis=1), yhat_valid)
classes = ['anger', 'disgust', 'fear', 'happiness', 'sadness', 'surprise',
'neutral']
sns.heatmap(cf_matrix, annot=True, xticklabels=classes,
yticklabels=classes)
```

Sample images along with their true and predicted labels are displayed to assess the model performance:

```
mapper = {0: 'anger', 1: 'disgust', 2: 'fear', 3: 'happiness', 4:
'sadness', 5: 'surprise', 6: 'neutral'}
np.random.seed(2)
random_sad_imgs = np.random.choice(np.where(y_valid[:, 1] == 1)[0], size=9)
random_neutral_imgs = np.random.choice(np.where(y_valid[:, 2] == 1)[0],
size=9
```

Speech Analysis

We start by obtaining the ‘toronto-emotional-speech-set-tess’ dataset for our speech analysis through the ‘kagglehub’ library.

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("ejlok1/toronto-emotional-speech-set-tess")

print("Path to dataset files:", path)
```

This part indicates the main directory where the dataset should be located on the local system.

```
base_dir = "/root/.cache/kagglehub/datasets/ejlok1/toronto-emotional-speech-set-tess/versions/1/TESS Toronto emotional speech set data"
```

Next, we import the necessary modules required for data manipulation, numerical and matrix computations, audio signal processing, data visualization and interacting with the operating system.

```
import pandas as pd
import numpy as np
import librosa
import matplotlib.pyplot as plt
import os
```

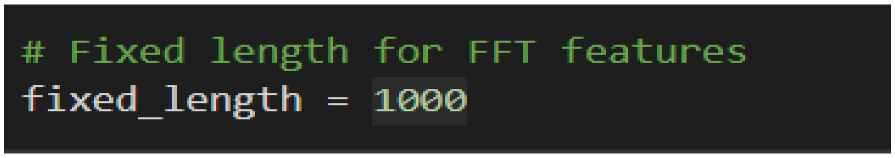
This function takes an audio file and extracts the features in the frequency domain. It first loads the audio file, performs a Fast Fourier Transform (FFT) to transform the time-domain signal into the frequency domain, calculates the magnitude spectrum of the result, which is the amplitude of each frequency component, and then extracts only the first half of the magnitude spectrum because the feature vector lengths might differ between different audio files. Finally, the feature vector is padded or trimmed to a fixed length using appropriate techniques to standardize the representation of the audio's frequency content.

```
def extract_fft_features(file_path, fixed_length):
    y, sr = librosa.load(file_path, sr=None)
    fft_result = np.fft.fft(y)
    magnitude_spectrum = np.abs(fft_result)
    # Use the first half of the FFT result
    features = magnitude_spectrum[:len(magnitude_spectrum) // 2]
    # Pad or trim the features to the fixed length
    if len(features) < fixed_length:
        features = np.pad(features, (0, fixed_length - len(features)), 'constant')
    else:
        features = features[:fixed_length]
    return features
```

An empty list is created and assigned to the variable ‘data’

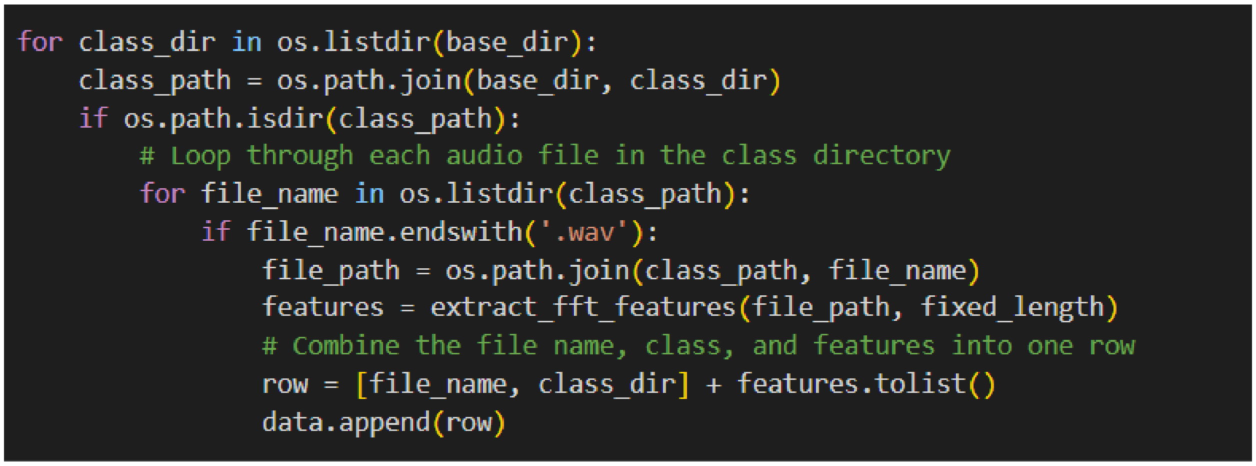
```
data = []
```

The line fixed_length = 1000 assigns a constant value of 1000 to control the length of the extracted FFT features so that the sizes of the feature vectors are consistent



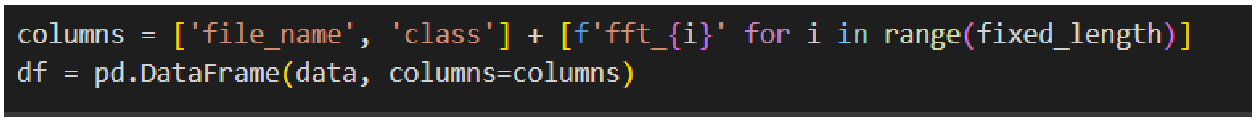
```
# Fixed length for FFT features
fixed_length = 1000
```

In the next step, the code iterates through a directory of .wav audio files, separated into classes, and iteratively calls **extract_fft_features** over each of the found files. Data now comprises the output lists, where each one will contain frequency-based features together with the file name and a class label.



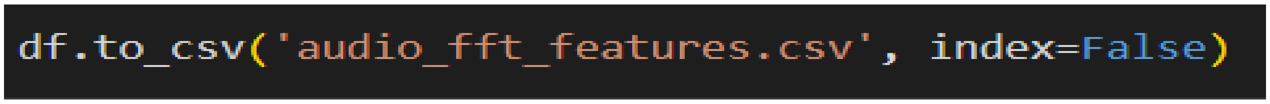
```
for class_dir in os.listdir(base_dir):
    class_path = os.path.join(base_dir, class_dir)
    if os.path.isdir(class_path):
        # Loop through each audio file in the class directory
        for file_name in os.listdir(class_path):
            if file_name.endswith('.wav'):
                file_path = os.path.join(class_path, file_name)
                features = extract_fft_features(file_path, fixed_length)
                # Combine the file name, class, and features into one row
                row = [file_name, class_dir] + features.tolist()
                data.append(row)
```

Next, the given code below creates a pandas data frame to correctly organize the extracted audio features. It defines column names for the file name, class label, and the extracted FFT features. Then it constructs the data frame using the extracted feature data assigning appropriate column names.



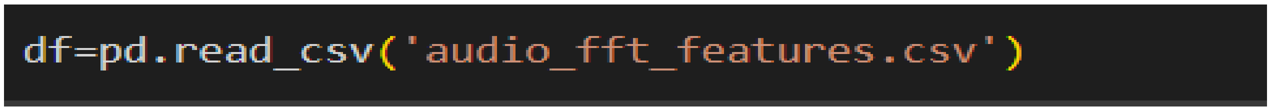
```
columns = ['file_name', 'class'] + [f'fft_{i}' for i in range(fixed_length)]
df = pd.DataFrame(data, columns=columns)
```

The command exports the data from a Pandas DataFrame to a CSV file titled **audio_fft_features.csv**, making sure that only the data is present, excluding the row indices by using the command **index = False**.



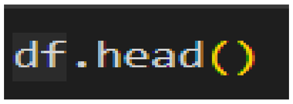
```
df.to_csv('audio_fft_features.csv', index=False)
```

Next line of code reads a CSV file named **'audio_fft_features.csv'** and loads its contents into a pandas DataFrame.



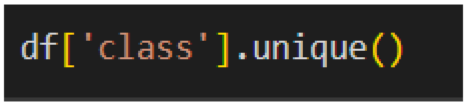
```
df=pd.read_csv('audio_fft_features.csv')
```

The head of the imported DataFrame is visualized using the command:



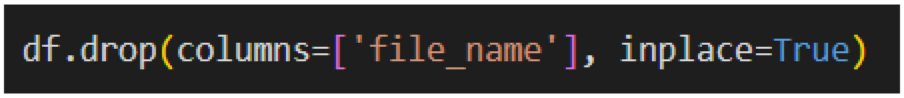
```
df.head()
```

This code line retrieves the distinct class labels found in the 'class' column of the pandas DataFrame. The unique() function of pandas series produces an array that includes solely the distinct values from the given column.



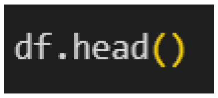
```
df['class'].unique()
```

This line of code, utilizing the pandas library, eliminates the column titled 'file_name' from the DataFrame. The inplace=True parameter changes the DataFrame directly, without generating a new copy.



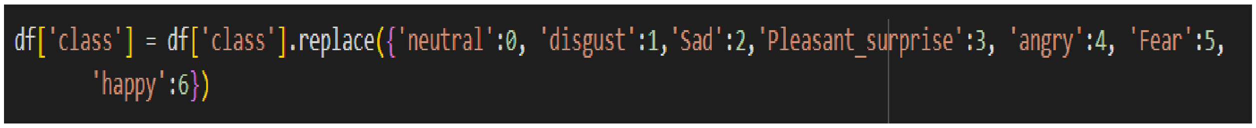
```
df.drop(columns=['file_name'], inplace=True)
```

This line again prints the head of the DataFrame.



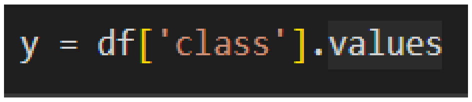
```
df.head()
```

This line of code substitutes the string labels in the 'class' column of the DataFrame 'df' with numeric values. This procedure is referred to as label encoding, which is frequently essential for machine learning algorithms that need numerical input. The code precisely assigns the string labels 'neutral', 'disgust', 'Sad', 'Pleasant_surprise', 'angry', 'Fear', and 'happy' to the integers 0 through 6, respectively.



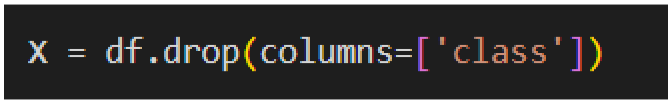
```
df['class'] = df['class'].replace({'neutral':0, 'disgust':1,'Sad':2,'Pleasant_surprise':3, 'angry':4, 'Fear':5,
       'happy':6})
```

This piece of code retrieves the values from the 'class' column of the DataFrame 'df' and saves them as a NumPy array in the variable 'y'.



```
y = df['class'].values
```

The next line of code creates a new DataFrame named 'X' by removing or dropping the 'class' column from the original DataFrame 'df'.

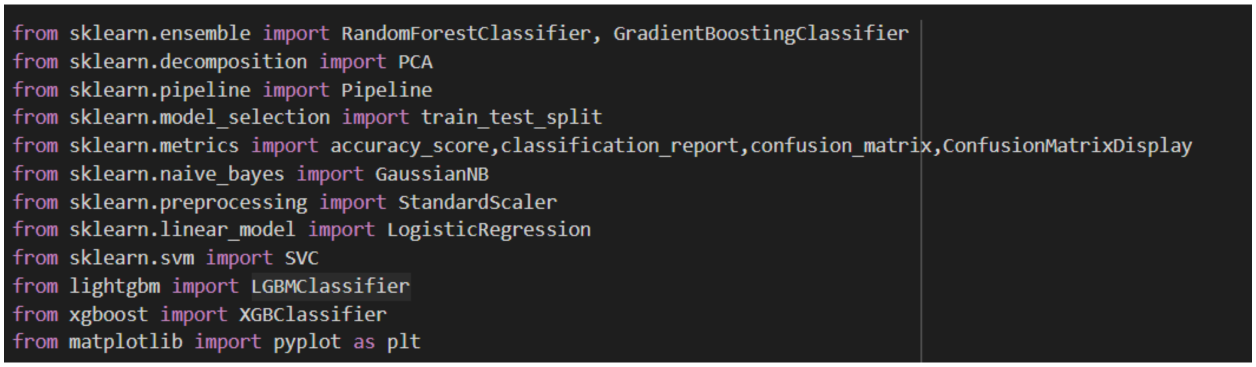


```
X = df.drop(columns=['class'])
```

This code snippet brings in different machine learning libraries, models, and utilities from well-known sources like scikit-learn, LightGBM, and XGBoost, along with visualization tools from matplotlib. These elements fulfill various roles:

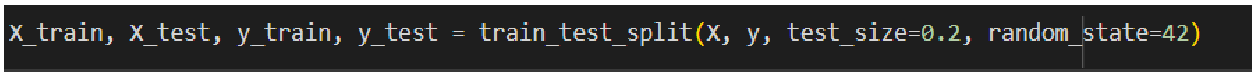
- Models and Classifiers: This encompasses machine learning algorithms like Random Forest, Gradient Boosting, Naive Bayes, Logistic Regression, Support Vector Classifier (SVC), LightGBM, and XGBoost that are utilized for constructing and training predictive models.
- Preprocessing and Dimensionality Reduction: Tools such as StandardScaler (to normalize data) and PCA (Principal Component Analysis, for decreasing feature count) assist in getting the data ready for modeling.
- Pipeline and Splitting: The Pipeline tool streamlines the process by linking several stages, whereas train_test_split is utilized to separate the dataset into training and testing portions.
- Evaluation Metrics: This encompasses techniques for assessing model performance, including accuracy, classification reports, confusion matrices, and visual representations of confusion matrices.

- Visualization: matplotlib.pyplot enables the creation of plots and graphs to enhance comprehension of data and the performance of models.



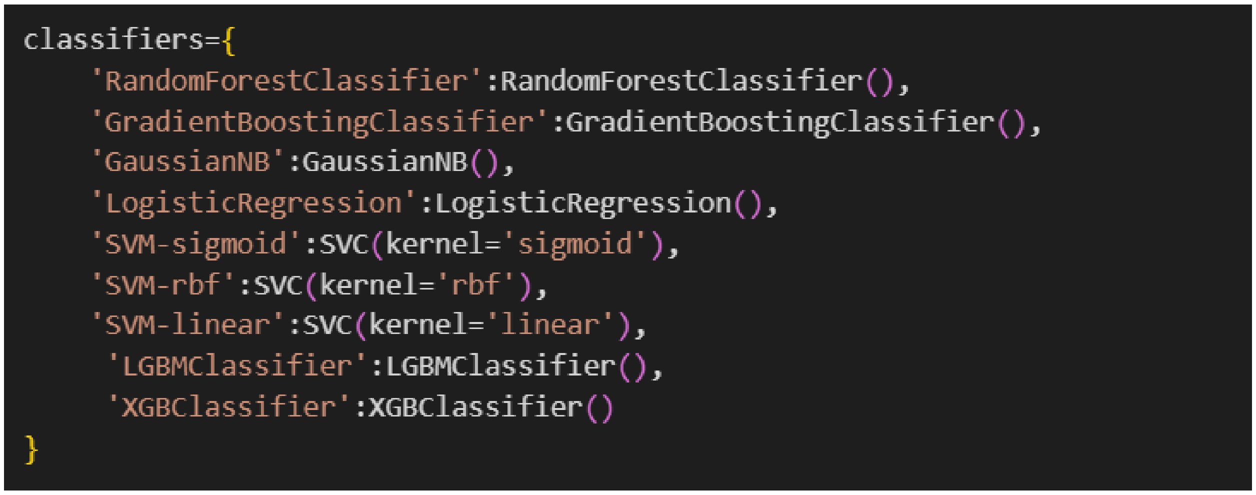
```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,classification_report,confusion_matrix,ConfusionMatrixDisplay
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from matplotlib import pyplot as plt
```

The dataset is then partitioned into training (80%) and testing (20%) sets for features (X) and labels (y), ensuring reproducibility with random_state=42



```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

This code creates a dictionary, `classifiers`, which links the names of different machine learning algorithms to their respective instantiated classifier objects. The collection features a varied assortment of models namely Random Forest, Gradient Boosting, LightGBM, XGBoost,Gaussian Naive Bayes, Logistic Regression, and Support Vector Machines utilizing sigmoid, rbf and linear kernel functions, enabling organized experiments and performance assessments in predictive modeling assignments.



```
classifiers={
    'RandomForestClassifier':RandomForestClassifier(),
    'GradientBoostingClassifier':GradientBoostingClassifier(),
    'GaussianNB':GaussianNB(),
    'LogisticRegression':LogisticRegression(),
    'SVM-sigmoid':SVC(kernel='sigmoid'),
    'SVM-rbf':SVC(kernel='rbf'),
    'SVM-linear':SVC(kernel='linear'),
     'LGBMClassifier':LGBMClassifier(),
     'XGBClassifier':XGBClassifier()
 }
```

This code trains, evaluates, and saves multiple machine learning models. For each model in the predefined dictionary, it trains the classifier using the training dataset and makes predictions on the test dataset. It evaluates performance using metrics such as accuracy, a classification report (including precision, recall, and F1-score), and a confusion matrix, which is also displayed visually. Finally, each trained model is saved as a file in the "saved_models" directory using `joblib`, enabling reuse without retraining. This approach streamlines the workflow for model comparison and deployment.

```
import joblib

for clf_name,clf in classifiers.items():
  print(f"Training and Evaluating {clf_name}")
  clf.fit(X_train,y_train)
  y_pred=clf.predict(X_test)
  print(f"{clf_name}Accuracy: {accuracy_score(y_test,y_pred)}")
  print(classification_report(y_test,y_pred))
  cm=confusion_matrix(y_test,y_pred)
  disp=ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=clf.classes_)
  disp.plot()
  plt.show()

  # Save the trained model
  model_path = os.path.join("saved_models", f"{clf_name}.pkl")
  joblib.dump(clf, model_path)
  print(f"Saved {clf_name} to {model_path}")
```

LSTM

A function called extract_features is formulated to analyze an audio file and extract important features. It imports the audio file and calculates features like chroma short-time Fourier transform (STFT), FFT average, MFCCs (Mel-frequency cepstral coefficients), and spectrogram average. These characteristics are compiled into one array for use in audio analysis and modeling tasks.

```
def extract_features(file_path):
    # Load audio file
    y, sr = librosa.load(file_path)

    # Extract features
    chroma_stft = librosa.feature.chroma_stft(y=y, sr=sr)
    chroma_stft_mean = np.mean(chroma_stft, axis=1)

    fft = np.fft.fft(y)
    fft_mean = np.mean(np.abs(fft))

    mfccs = librosa.feature.mfcc(y=y, sr=sr)
    mfccs_mean = np.mean(mfccs, axis=1)

    spectrogram = librosa.amplitude_to_db(librosa.stft(y))
    spectrogram_mean = np.mean(spectrogram, axis=1)

    # Concatenate features
    features = np.concatenate((chroma_stft_mean, [fft_mean], mfccs_mean,
spectrogram_mean))

    return  chroma_stft,fft_mean,mfccs_mean,spectrogram_mean
```

An empty list is created and assigned to the variable ‘data’

```
data =[]
```

This line of code imports necessary libraries: os for file system operations, librosa for audio processing, numpy for numerical computations, and pandas for data manipulation.

```
import os
import librosa
import numpy as np
import pandas as pd
```

This code snippet iterates over a directory tree of audio files for each class. For every.WAV file it encounters it extracts a set of audio features using the extract_features method. These features include chroma STFT, mean of FFT, mean of MFCCs, and mean of spectrogram.These features, along with the filename and class label, are then appended to a list named data for further processing.

```
for class_dir in os.listdir(base_dir):
    class_path = os.path.join(base_dir, class_dir)
    if os.path.isdir(class_path):
        # Loop through each audio file in the class directory
        for file_name in os.listdir(class_path):
            if file_name.endswith('.wav'):
                file_path = os.path.join(class_path, file_name)
                chroma_stft,fft_mean,mfccs_mean,spectrogram_mean =
extract_features(file_path)
                # Combine the file name, class, and features into one row
                row = [file_name, class_dir,
chroma_stft,fft_mean,mfccs_mean,spectrogram_mean]
                data.append(row)
```

This code generates a pandas DataFrame to hold the extracted audio characteristics. It specifies column names and builds the DataFrame with the gathered feature data, allocating the relevant column names.

```
columns = ['file_name', 'class','chroma_stft_mean', 'fft_mean', 'mfccs_mean',
'spectrogram_mean']
df = pd.DataFrame(data, columns=columns)
```

This line of code displays the first 5 rows of the DataFrame ‘df’. It helps to get a visual overview of the data and check if it has been loaded and processed correctly.

```
df.head()
```

This line of code prints the length of the array located at the 145th index of the 'chroma_stft_mean' column in the DataFrame 'df'.

```
print(len(df["chroma_stft_mean"][145]))
```

This code loops through designated columns ('mfccs_mean', 'chroma_stft_mean', 'spectrogram_mean') within the DataFrame. For every column, a function (lambda x: np.array(x).flatten()) is applied to every element. This function transforms each element into a NumPy array and subsequently compresses it into a one-dimensional array. This guarantees a uniform data format for future processing.

```
for col in ['mfccs_mean', 'chroma_stft_mean', 'spectrogram_mean']:
    df[col] = df[col].apply(lambda x: np.array(x).flatten())
```

This line of code finds the maximum length of arrays in column 'chroma stft mean' of DataFrame ‘df’.

```
max_length = max(len(arr) for arr in df['chroma_stft_mean'])
```

Next code snippet creates a new DataFrame **df_flattened** by horizontally concatenating several DataFrames. First, it drops the original columns 'chroma_stft_mean', 'mfccs_mean', and 'spectrogram_mean' from the original DataFrame. Then it creates new DataFrames for each of these features, in which each element of the original list-like column is turned into separate columns with corresponding names. So, nested list structures contained in these columns are flattened, and the data is prepared for further analysis or training of machine learning models.

```
df_flattened = pd.concat([
    df.drop(['chroma_stft_mean', 'mfccs_mean', 'spectrogram_mean'], axis=1),
   pd.DataFrame(df['chroma_stft_mean'].tolist(),
columns=[f'chroma_stft_mean_{i}' for i in range(max_length)]),
    pd.DataFrame(df['mfccs_mean'].tolist(), columns=[f'mfccs_mean_{i}' for i in
range(len(df['mfccs_mean'].iloc[0]))]),
    pd.DataFrame(df['spectrogram_mean'].tolist(),
columns=[f'spectrogram_mean_{i}' for i in
range(len(df['spectrogram_mean'].iloc[0]))])
], axis=1)
```

This line replaces all missing values in the DataFrame **df_flattened** with zeros.

```
df_flattened.fillna(0, inplace=True)
```

This code calculates the total number of rows in the DataFrame `df_flattened` that contain at least one missing value.

```
df_flattened.isna().any(axis=1).sum()
```

This line of code displays the first 5 rows of the DataFrame `df_flattened`.

```
df_flattened.head()
```

This replaces the categorical class labels within the class column of the dataframe df with numerical values, where each emotion will be represented by its unique integer.

```
df['class'] = df['class'].replace({'neutral':0,
'disgust':1,'Sad':2,'Pleasant_surprise':3, 'angry':4, 'Fear':5,'happy':6})
```

This code imports essential libraries and tools for data processing and building a machine learning model. It uses **Pandas** and **NumPy** for data manipulation and analysis, along with tools like **LabelEncoder** for encoding categorical data, **StandardScaler** for normalizing data, and **train_test_split** for dividing datasets. It also imports TensorFlow and its Keras API to construct a sequential deep learning model. The layers include **LSTM** (for sequential data processing), **Dense** (fully connected layers for prediction), and **Dropout** (to prevent overfitting). These components are typically used for developing and training neural networks, particularly for time-series or sequence-based tasks.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
```

This code separates the features and labels from the DataFrame `df_flattened`. The features (`X`) are extracted by dropping the 'file_name' and 'class' columns, and converting the remaining data into a NumPy array. The **labels (`y`)** are extracted from the 'class' column, which represents the target variable for classification. These arrays are prepared for use in training a machine learning model.

```
X = df_flattened.drop(columns=['file_name', 'class']).values
y = df_flattened['class'].values
```

The present code utilizes LabelEncoder to convert the target labels (y) into numerical values, effectively transforming categorical labels into a format that is appropriate for machine learning algorithms.

```
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
```

This code standardizes the feature data (X) by scaling to give a mean of 0 and a standard deviation of 1, due to StandardScaler, which helps the model improve in performance by ensuring consistent feature scaling.

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

This transforms feature data (`X`) into a 3D array which is exactly what is required to feed to an LSTM model.

```
X = X.reshape((X.shape[0], 1, X.shape[1]))
```

This code partitions our dataset into training (80%) and testing (20%) sets for features (X) and labels (y), ensuring reproducibility with random_state=42

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

This code defines a Sequential LSTM model for classification. It starts with an LSTM layer of 128 units, specifying the input shape of the data and setting `return_sequences=True` to output sequences for the next LSTM layer. Then it adds a dropout layer with a 20% dropout rate, followed by another LSTM layer with 64 units. Another dropout layer is added followed by a Dense layer with 32 units and ReLU activation function. Finally, the output layer has a number of units equal to the unique classes in the target variable (`y`) with softmax activation to produce probabilities for each class.

```
lstm_model = Sequential()
lstm_model.add(LSTM(128, input_shape=(X_train.shape[1], X_train.shape[2]),
return_sequences=True))
lstm_model.add(Dropout(0.2))
lstm_model.add(LSTM(64))
lstm_model.add(Dropout(0.2))
lstm_model.add(Dense(32, activation='relu'))
lstm_model.add(Dense(len(np.unique(y)), activation='softmax'))
```

The model is then compiled using the Adam optimizer and sparse categorical cross-entropy loss function, with accuracy as the metric. It is then trained for 50 epochs at a batch size of 32, using 20% of the data for validation. At the end of training, the model is saved as a `.keras` file for later use.

```
lstm_model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy']) history =
lstm_model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2) lstm_model.save('lstm_model.keras')
```

In this code line, the model makes predictions on the test data (`X_test`), which then have their predicted probabilities converted into class labels using `argmax`. That will select the highest probability class for each sample.

```
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Predict the labels for the test set
y_pred = lstm_model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
```

This code prints the classification report, which computes detailed metrics, such as precision, recall, and F1-score, for each class in the target variable; it is computed based on the true labels (`y_test`) and predicted labels (`y_pred_classes`). Further, it computes the confusion matrix, a way of measuring the performance of a model. In this confusion matrix, one can view the counts for true positives, false positives, true negatives, and false negatives. This confusion matrix is visualized using a heatmap, where each cell is annotated with the corresponding count, and the axis labels are the class names, providing an intuitive view of the model's performance across different classes.

```
print("Classification Report:")
print(classification_report(y_test, y_pred_classes,
target_names=label_encoder.classes_))

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
```

```
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

MFCC

The code defines a function called extract_features that extracts the key audio features from an audio file, such as chroma STFT, FFT mean, MFCCs, and spectrogram mean. These features are calculated by processing the audio data, where each feature is summarized by its mean value. The extracted features are then concatenated into a single array that can be used for further analysis or modeling. The variable data = [] initializes an empty list that will probably be used to store the extracted features for multiple audio files.

```
def extract_features(file_path):
    # Load audio file
    y, sr = librosa.load(file_path)

    # Extract features
    chroma_stft = librosa.feature.chroma_stft(y=y, sr=sr)
    chroma_stft_mean = np.mean(chroma_stft, axis=1)

    fft = np.fft.fft(y)
    fft_mean = np.mean(np.abs(fft))

    mfccs = librosa.feature.mfcc(y=y, sr=sr)
    mfccs_mean = np.mean(mfccs, axis=1)

    spectrogram = librosa.amplitude_to_db(librosa.stft(y))
    spectrogram_mean = np.mean(spectrogram, axis=1)

    # Concatenate features
    features = np.concatenate((chroma_stft_mean, [fft_mean], mfccs_mean, 
spectrogram_mean))

    return  chroma_stft,fft_mean,mfccs_mean,spectrogram_mean
data =[]
```

This code loads the necessary libraries and modules processes a directory structure where each subdirectory represents a class, and it contains.wav audio files. For every audio file, it gets features such as chroma STFT, FFT mean, MFCCs mean, and spectrogram mean using the extract_features function. All these results are combined into a row with the file name, class label, and extracted features. These rows are added to the data list, which contains features for all audio files across all classes. The data can then be used for model training or analysis.

```python
import os
import librosa
import numpy as np
import pandas as pd

for class_dir in os.listdir(base_dir):
    class_path = os.path.join(base_dir, class_dir)
    if os.path.isdir(class_path):
        # Loop through each audio file in the class directory
        for file_name in os.listdir(class_path):
            if file_name.endswith('.wav'):
                file_path = os.path.join(class_path, file_name)
                chroma_stft,fft_mean,mfccs_mean,spectrogram_mean =
extract_features(file_path)
                # Combine the file name, class, and features into one row
                row = [file_name, class_dir,
chroma_stft,fft_mean,mfccs_mean,spectrogram_mean]
                data.append(row)
```

This code forms a DataFrame using the list `data`. The code flattens feature arrays (`mfccs_mean`, `chroma_stft_mean`, `spectrogram_mean`) to create one-dimensional arrays, calculates the max length of `chroma_stft_mean` feature arrays and declares a variable max_length where the longest vector will be placed.

```python
columns = ['file_name', 'class','chroma_stft_mean', 'fft_mean',
'mfccs_mean', 'spectrogram_mean']
df = pd.DataFrame(data, columns=columns) for col in ['mfccs_mean',
'chroma_stft_mean', 'spectrogram_mean']:
df[col] = df[col].apply(lambda x: np.array(x).flatten())
max_length = max(len(arr) for arr in df['chroma_stft_mean'])
```

This code creates a new DataFrame `df_m` from `df` by selecting the `class` column and transforming the `mfccs_mean` feature, which is a list, into separate columns, one for each value in the `mfccs_mean` array. Missing values in `df_m` are replaced with zeros using `fillna(0)`. Then, the `class` column in the original `df` is replaced with numerical labels corresponding to different emotion categories (for example, 'neutral' becomes 0, 'disgust' becomes 1, etc.), and it becomes ready for machine learning models.

```python
df_m= pd.concat([df["class"], pd.DataFrame(df['mfccs_mean'].tolist(),
columns=[f'mfccs_mean_{i}' for i in
range(len(df['mfccs_mean'].iloc[0]))])], axis=1)
df_m.fillna(0, inplace=True)
df['class'] = df['class'].replace({'neutral':0,
'disgust':1,'Sad':2,'Pleasant_surprise':3, 'angry':4, 'Fear':5, 'happy':6})
```

The code extracts the target labels (`y`) out from the `class` column and saves the features by getting rid of the `class` column in `df_m`.

```
y = df['class'].values
X = df_m.drop(columns=['class']).values
```

This code imports the necessary tools required for modelling and applies label encoding to the target variable `y`, which transforms categorical labels into numerical values using `LabelEncoder`. This is usually done before training machine learning models

```
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout
import joblib

label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
```

This code snippet saves the label encoder as a file named `label_encoder.pkl` and the standard scaler as a file named `standard_scaler.pkl` using `joblib`. It also standardizes the feature data `X` by scaling it to have a mean of 0 and a standard deviation of 1 using `StandardScaler`.

```
joblib.dump(label_encoder, 'label_encoder.pkl')
scaler = StandardScaler()
X = scaler.fit_transform(X)
joblib.dump(scaler, 'standard_scaler.pkl')
```

Here, feature data `X` is reshaped into a 3D array, which is required for LSTM models (samples, timesteps, features). Then, it splits the data into training and testing sets, with 80% for training and 20% for testing, using `train_test_split`.

```
X = X.reshape((X.shape[0], 1, X.shape[1]))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

This code defines and trains an LSTM down to prevent overfitting with two LSTM layers. The final set of dense layers outputted using the `softmax` activation function will output a class probability. The models will be compiled with the Adam Optimizer and sparse categorical cross entropy loss. Then, it feeds these models into the actual training data for 50 epochs with a batch size of 32 and splits 20% for validation.

```
model = Sequential()
model.add(LSTM(128, input_shape=(X_train.shape[1], X_train.shape[2]),
return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(64))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(len(np.unique(y)), activation='softmax'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2)
```

This code uses a trained model to predict labels for the test set, makes a classification report that may include precision, recall, and F1-score metrics, calculates the confusion matrix, and visualizes the heatmap of the relationship between the actual and predicted labels.

```
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Predict the labels for the test set
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
print("Classification Report:")
# Convert label_encoder.classes_ to a list of strings
target_names = [str(cls) for cls in label_encoder.classes_]
print(classification_report(y_test, y_pred_classes, target_names=target_names))

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=target_names, yticklabels=target_names) # Use target_names here as
well
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

The trained model is saved for future use.

```
model.save('mfcc_lstm_model.keras')
```

Spectogram

The function extract_features is designed to load an audio file and extract various audio features which can be used for further audio analysis, such as classification tasks.

```
import librosa
import numpy as np

def extract_features(file_path):
    # Load audio file
    y, sr = librosa.load(file_path)
```

Loading the Audio File: The librosa.load(file_path) function loads an audio file from the specified path. The returned variables y and sr contain the time series data of the audio and the sample rate, respectively.

```
chroma_stft = librosa.feature.chroma_stft(y=y, sr=sr)
chroma_stft_mean = np.mean(chroma_stft, axis=1)
```

Chroma STFT: This feature represents the energy distribution across 12 pitch classes. The librosa.feature.chroma_stft function computes the chromagram from the Short-Time Fourier Transform (STFT) of the audio signal. By taking the mean of these chroma features across time, we get a summary statistic (chroma_stft_mean) that captures the average pitch content.

```
# Compute FFT
fft = np.fft.fft(y)
fft_mean = np.mean(np.abs(fft))
```

FFT (Fast Fourier Transform): The FFT converts the time domain signal into the frequency domain. By calculating the mean of the absolute values of the FFT, we get a single value (fft_mean) that represents the average energy across the frequency spectrum.

```
# Extract MFCCs
mfccs = librosa.feature.mfcc(y=y, sr=sr)
mfccs_mean = np.mean(mfccs, axis=1)
```

MFCCs (Mel-Frequency Cepstral Coefficients): MFCCs are widely used in audio signal processing and are a representation of the short-term power spectrum of a sound. The librosa.feature.mfcc function computes these coefficients, and their mean values across time (mfccs_mean) provide a summary statistic of the spectral characteristics.

```
# Compute Spectrogram
spectrogram = librosa.amplitude_to_db(librosa.stft(y))
spectrogram_mean = np.mean(spectrogram, axis=1)
```

Spectrogram: This feature represents the amplitude of frequencies over time. The Short-Time Fourier Transform (STFT) of the signal is computed, and then converted to decibel scale using `librosa.amplitude_to_db`. The mean values of the spectrogram (`spectrogram_mean`) provide an average representation of the frequency content over time.

```
# Concatenate features
features = np.concatenate((chroma_stft_mean, [fft_mean], mfccs_mean,
spectrogram_mean))
return chroma_stft, fft_mean, mfccs_mean, spectrogram_mean
```

Concatenation: All the extracted features are concatenated into a single feature vector, which includes the mean chroma STFT, mean FFT, mean MFCCs, and mean spectrogram values. The function returns these features for further processing.

The next part of the code prepares the data for modeling. It iterates through directories containing audio files, extracts features, and stores them in a pandas DataFrame.

```
data = []
base_dir = 'path_to_audio_files'
for class_dir in os.listdir(base_dir):
    class_path = os.path.join(base_dir, class_dir)
    if os.path.isdir(class_path):
        for file_name in os.listdir(class_path):
            if file_name.endswith('.wav'):
                file_path = os.path.join(class_path, file_name)
                chroma_stft, fft_mean, mfccs_mean, spectrogram_mean =
extract_features(file_path)
                row = [file_name, class_dir, chroma_stft, fft_mean,
mfccs_mean, spectrogram_mean]
                data.append(row)
columns = ['file_name', 'class', 'chroma_stft_mean', 'fft_mean',
'mfccs_mean', 'spectrogram_mean']
df = pd.DataFrame(data, columns=columns)
```

Directory Iteration and Feature Extraction: The code traverses through each class directory in the base directory, processes each `.wav` file, and extracts the features using the `extract_features` function. The extracted features, along with the file name and class label, are appended to a list, which is then converted to a DataFrame.

The extracted features are flattened, the class labels are encoded, and the feature matrix is prepared for model training.

```
for col in ['mfccs_mean', 'chroma_stft_mean', 'spectrogram_mean']:
    df[col] = df[col].apply(lambda x: np.array(x).flatten())
df['class'] = df['class'].replace({'neutral': 0, 'disgust': 1, 'Sad': 2,
'Pleasant_surprise': 3, 'angry': 4, 'Fear': 5, 'happy': 6})
df_spect = pd.concat([df["class"],
pd.DataFrame(df['spectrogram_mean'].tolist(),
columns=[f'spectrogram_mean_{i}' for i in
range(len(df['spectrogram_mean'].iloc[0]))])], axis=1)
df_spect.fillna(0, inplace=True)
```

Flattening and Encoding: The feature columns (`mfccs_mean`, `chroma_stft_mean`, `spectrogram_mean`) are flattened. The class labels are replaced with numerical values. The DataFrame is then prepared for the spectrogram features, ensuring all entries have the same length by filling missing values with zeros.

The data is split into training and testing sets, scaled, and reshaped for the LSTM model.

```
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout
y = df_spect['class'].values
X = df_spect.drop(columns=['class']).values
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
scaler = StandardScaler()
X = scaler.fit_transform(X)
X = X.reshape((X.shape[0], 1, X.shape[1]))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Label Encoding and Scaling: The class labels are encoded using `LabelEncoder`, and the features are scaled using `StandardScaler`. The feature matrix is reshaped to match the input requirements of the LSTM model.

An LSTM model is defined, compiled, and trained using Keras.

```
model = Sequential()
model.add(LSTM(128, input_shape=(X_train.shape[1], X_train.shape[2]),
return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(64))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(len(np.unique(y)), activation='softmax'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2)
```

The model includes two LSTM layers with dropout regularization to prevent overfitting, followed by dense layers. The final layer uses a softmax activation function for classification.

The model's performance is evaluated using a classification report and a confusion matrix.

```
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
print("Classification Report:")
target_names = [str(cls) for cls in label_encoder.classes_]
print(classification_report(y_test, y_pred_classes,
target_names=target_names))
conf_matrix = confusion_matrix(y_test, y_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=target_names, yticklabels=target_names)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

The classification report provides precision, recall, and F1 scores for each class. The confusion matrix visualizes the model's performance, showing the counts of true vs. predicted labels.

Finally, the trained model is saved for future use.

```
model.save('spectogram_lstm_model.keras')
```

References:

Anshu, Malhotra., Rajni, Jindal. (2020). Multimodal Deep Learning based Framework for Detecting Depression and Suicidal Behaviour by Affective Analysis of Social Media Posts. EAI Endorsed Transactions on Pervasive Health and Technology, doi: 10.4108/EAI.13-7-2018.164259

Diana, Ramírez-Cifuentes., Ana, Freire., Ricardo, Baeza-Yates., Joaquim, Puntí., Pilar, Medina-Bravo., Diego, Velázquez., Josep, M., Gonfaus., Jordi, Gonzàlez. (2020). Detection of Suicidal Ideation on Social Media: Multimodal, Relational, and Behavioral Analysis.. Journal of Medical Internet Research, doi: 10.2196/17758

Joshua, Cohen., Vanessa, Richter., Michael, Neumann., David, P., Black., Allie, Haq., Jennifer, Wright-Berryman., V., Ramanarayanan. (2023). A multimodal dialog approach to mental state characterization in clinically depressed, anxious, and suicidal populations. Frontiers in Psychology, doi: 10.3389/fpsyg.2023.1135469

Moumita, Chatterjee., Piyush, Kumar., Poulomi, Samanta., Dhrubasish, Sarkar. (2022). Suicide ideation detection from online social media: A multi-modal feature based technique. International journal of information management data insights, doi: 10.1016/j.jjimei.2022.100103

Pradeep, Kumar., Dilip, Singh, Sisodia., Rahul, Shrivastava. (2024). A Deep Learning-Based Sentiment Classification Approach for Detecting Suicidal Ideation on Social Media Posts. Communications in computer and information science, doi: 10.1007/978-3-031-54547-4_21

Fahim K. Sufi; Ibrahim Khalil (2022). Automated Disaster Monitoring from Social Media Posts using AI based Location Intelligence and Sentiment Analysis. doi: 10.36227/techrxiv.19212105.v1

Barua, P.D., Vicnesh, J., Lih, O.S. et al. (2024). Artificial intelligence assisted tools for the detection of anxiety and depression leading to suicidal ideation in adolescents: a review. Cogn Neurodyn 18, 1–22 https://doi.org/10.1007/s11571-022-09904-0

I. J. Goodfellow, D. Erhan, P. L. Carrier, A. Courville, M. Mirza, B. Hamner, W. Cukierski, Y. Tang, D. Thaler, D.-H. Lee, Y. Zhou, C. Ramaiah, F. Feng, R. Li, X. Wang, D. Athanasakis, J. Shawe-Taylor, M. Milakov, J. Park, R. Ionescu, M. Popescu, C. Grozea, J. Bergstra, J. Xie, L. Romaszko, B. Xu, Z. Chuang, and Y. Bengio. Challenges in representation learning: A report on three machine learning contests. Neural Networks, 64:59--63, 2015. Special Issue on "Deep Learning of Representations"

Pichora-Fuller, M. Kathleen; Dupuis, Kate, (2020). Toronto emotional speech set (TESS). , https://doi.org/10.5683/SP2/E8H2MF, Borealis, V1

Saif Mohammad and Felipe Bravo-Marquez. (2017). WASSA-2017 Shared Task on Emotion Intensity. In Proceedings of the 8th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis, pages 34–49, Copenhagen, Denmark. Association for Computational Linguistics.

https://www.geeksforgeeks.org/vgg-net-architecture-explained/

Sohaib & Ming, Zhao & Tang, Fengxiao & Zhu, Yusen. (2023). LWSE: a lightweight stacked ensemble model for accurate detection of multiple chest infectious diseases including COVID-19. Multimedia Tools and Applications. 83. 1-37. 10.1007/s11042-023-16432-4.

Tiwari, V., 2010. MFCC and its applications in speaker recognition. International journal on emerging technologies, 1(1), pp.19-22.

Wyse, L., 2017. Audio spectrogram representations for processing with convolutional neural networks. arXiv preprint arXiv:1706.09559.