Speech Emotion Recognition using Machine Learning

PRESENTED TO:

Dr.Gamal EL-Behery

PRESENTED BY:

1.	Esraa Ali Yousef Elmorsy	– ID: - 804610251
2.	Tasbeh Khaled elzakzouk	– ID: - 804620351
3.	Norhan Elsaid Helal	– ID: - 804614695
4.	Sara Shaaban Nasr Eldin Elshaer	– ID: - 804611546
5.	Rahma Mohamed Abo Shaheen	– ID: - 804611059
6.	Mariam Mohamed Abo elnour	– ID: - 804614089
7.	Renad Mohamed Sadat Alalafy	– ID: - 804611261
8.	Samia Nabil Mohamed Altonamly	– ID: - 804611867
9.	Maryam Mohamed Elghadban	– ID: - 804613988
10.	Norhan Hassan Ahmed	– ID: - 804639642
11.	Hanan Amer Abo Abdo	– ID: - 804621462
12.	Marwa Abdo Elabd	– ID: - 804636814

Table of Contents

1.Abstract

The project explores machine learning techniques for audio emotion detection. Leveraging a structured dataset of labeled audio files, the workflow incorporates feature extraction using Mel-Frequency Cepstral Coefficients (MFCCs), data preprocessing, and classification using algorithms such as **Random Forest, SVM**.The results highlight key insights into model performance and potential for real-world applications in fields like sentiment analysis, customer service, and entertainment.

2.Introduction

- **Background**: Emotion recognition in speech is critical in human-computer interaction. Audio signals provide rich data for interpreting emotions, which can be valuable in creating adaptive and intelligent systems.
- **Problem Statement**: Develop a model to classify audio signals into distinct emotional categories.
- **Objectives**: Extract meaningful features from audio files, build a robust classification model, and evaluate performance using real-world metrics.
- **Scope**: Focuses on supervised learning methods using labeled datasets.

Project GitHub Link:

https://github.com/rahma-mohmed/Machine-Learning-Project

3.Data Set

Data Source: https://zenodo.org/records/1188976

Key Attributes:

- Filenames encode emotion, intensity, and actor details.
- Example Filename: 03-01-06-01-02-01-12.wav.

Filename identifiers

- Modality (01 = full-AV, 02 = video-only, 03 = audio-only).
- Vocal channel (01 = speech, 02 = song).
- Emotion (01 = neutral, 02 = calm, 03 = happy, 04 = sad, 05 = angry, 06 = fearful, 07 = disgust, 08 = surprised).
- Emotional intensity (01 = normal, 02 = strong). NOTE: There is no strong intensity for the 'neutral' emotion.
- Statement (01 = "Kids are talking by the door", 02 = "Dogs are sitting by the door").
- Repetition (01 = 1st repetition, 02 = 2nd repetition).
- Actor (01 to 24. Odd numbered actors are male, even numbered actors are female).

4.Needed libraries For Project

Libraries used: librosa, pandas, numpy, matplotlib.pyplot, seaborn, sklearn (with specific modules like sklearn.preprocessing, sklearn.model_selection, sklearn.ensemble, and sklearn.metrics)

- pandas: Used for data **manipulation** and **analysis**, particularly for handling tabular data such as filenames, labels, and metadata.
- librosa: A Python library for audio and **music** analysis, utilized for extracting features like **MFCCs** from audio signals.
- numpy: Provides support for numerical operations and efficient handling of arrays, crucial for processing audio features.
- matplotlib.pyplot: A plotting library for creating visualizations, such as **spectrograms** and feature distributions.
- seaborn: Built on top of matplotlib, used for creating advanced and attractive statistical plots, like heatmaps for confusion matrices.
- sklearn.preprocessing: Provides tools for scaling and encoding data to prepare it for modeling.
- sklearn.model_selection: Includes methods like train_test_split for splitting datasets into **training** and **testing** subsets.
- sklearn. ensemble: Contains ensemble-based methods such as Random Forest used for **classification**.
- sklearn. metrics: Offers evaluation tools like classification_report and **confusion_matrix** to assess model performance.

5.Data Exploration

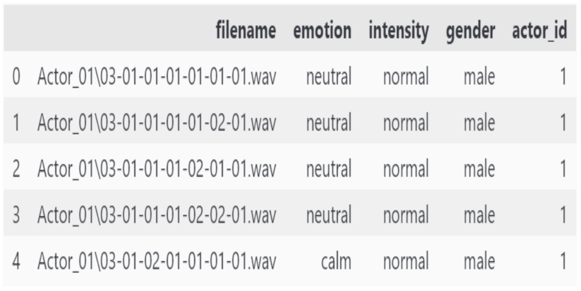
In this phase, the goal is to Understanding the Dataset, Review column names, types of data (categorical, numerical, etc.), and dataset size (number of rows/columns), Understand the distribution of each feature. A clear understanding of what the data represents and what insights or predictions it can support.

```
print('Size of Data: ', metadata.shape)

print("Metadata Sample:")

display(metadata.head())
```

1.check the shape

of the dataset, which gives me the number of rows and columns. The rows represent individual data records, and the columns represent the dataset's features.

output: Size of Data: (1440, 5)



	filename	emotion	intensity	gender	actor_id
0	Actor_01\03-01-01-01-01-01-01.wav	neutral	normal	male	1
1	Actor_01\03-01-01-01-01-02-01.wav	neutral	normal	male	1
2	Actor_01\03-01-01-01-02-01-01.wav	neutral	normal	male	1
3	Actor_01\03-01-01-01-02-02-01.wav	neutral	normal	male	1
4	Actor_01\03-01-02-01-01-01-01.wav	calm	normal	male	1

the features are:

file_name: The name of the file associated with each record.

emotion: The emotional label assigned to the data.

intensity: The intensity level of the emotion.

gender: The gender of the actor.

actor_id: A unique identifier for the actor.

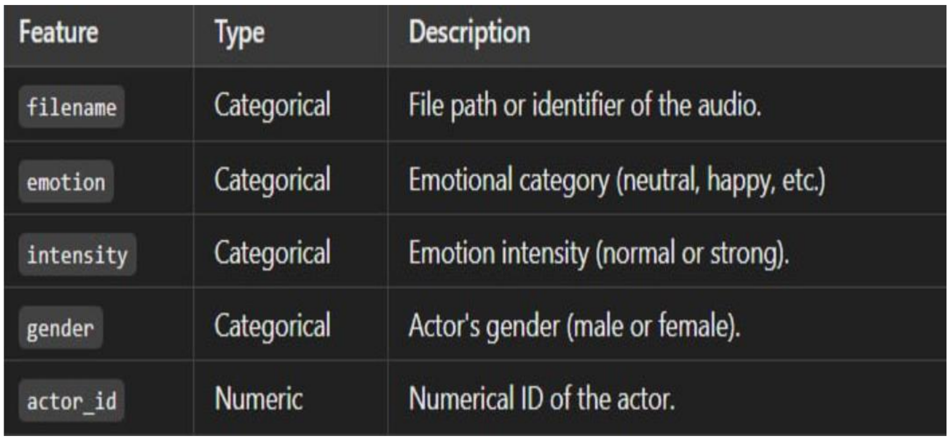
2.dtypes:

attribute to determine the datatype of each feature in my dataset. This helps me identify whether a feature is numerical, categorical, or textual.

```
print ('Data Types: ', metadata. Types)
```

Output

```
filename    object
emotion     object
intensity   object
gender      object
actor_id    int64
```



Feature	Type	Description
filename	Categorical	File path or identifier of the audio.
emotion	Categorical	Emotional category (neutral, happy, etc.)
intensity	Categorical	Emotion intensity (normal or strong).
gender	Categorical	Actor's gender (male or female).
actor_id	Numeric	Numerical ID of the actor.

```
metadata.isnull().sum()
```

3. isnull:

attribute to determine the if there is a null values in data.

output:

```
filename    0
emotion     0
intensity   0
gender      0
actor_id    0
```

4. check for unique values

```python
labels = metadata['emotion']
print("\nLabel Overview:")
print(f"Unique Emotion Labels: {labels.nunique()}")
print(f"Label Categories: {labels.unique()}")

labels = metadata['intensity']
print("\nLabel Overview:")
print(f"Unique Intensity Labels: {labels.nunique()}")
print(f"Label Categories: {labels.unique()}")

labels = metadata['gender']
print("\nLabel Overview:")
print(f"Unique gender Labels: {labels.nunique()}")
print(f"Label Categories: {labels.unique()}")
```

output :

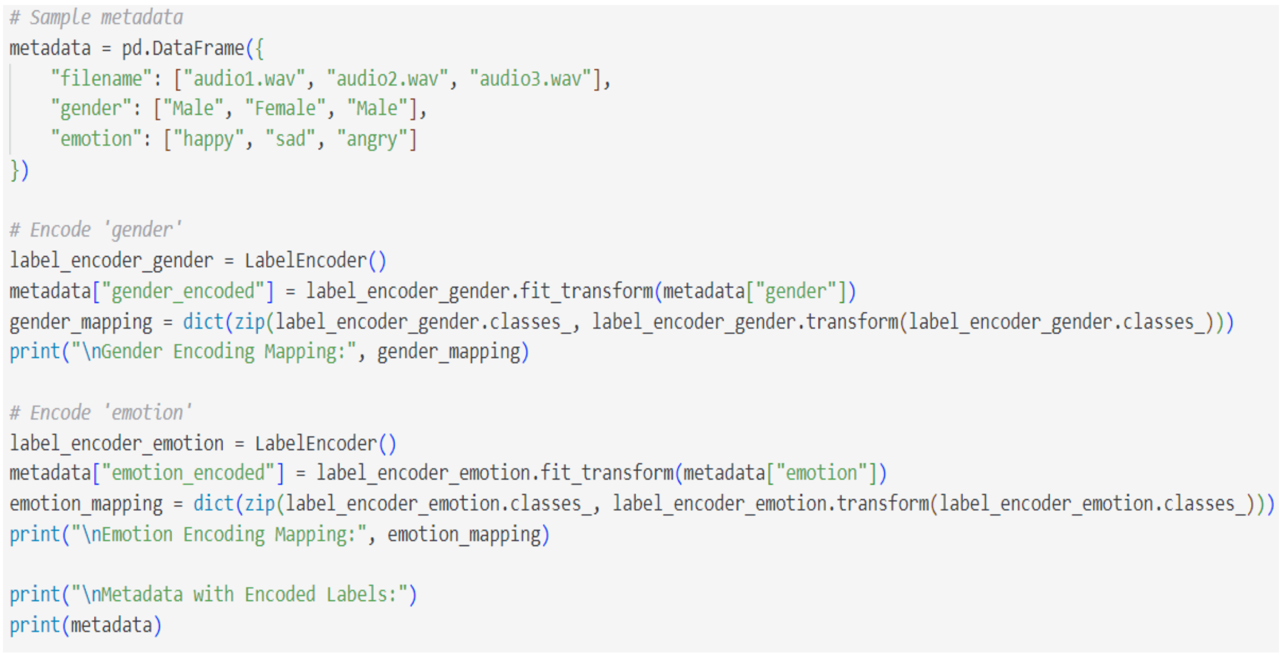
We Check unique Values We Have Unique Emotion Labels: 8

- Label Categories: ['neutral' 'calm' 'happy' 'sad' 'angry' 'fearful' 'disgust' 'surprised']
- and Unique Intensity Labels: 2
- Label Categories: ['normal' 'strong']
- Unique gender Labels: 2
- Label Categories: ['male' 'female']

6.Data Pre-Processing

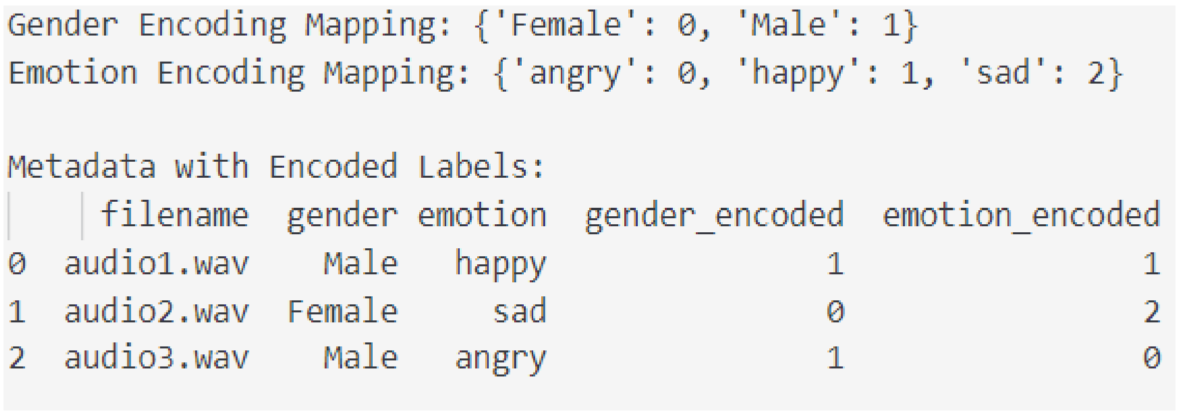
1-Label Encoding

for 'Gender' and 'Emotion' this helps to convert categoric data into numeric for calculations.



```python
# Sample metadata
metadata = pd.DataFrame({
    "filename": ["audio1.wav", "audio2.wav", "audio3.wav"],
    "gender": ["Male", "Female", "Male"],
    "emotion": ["happy", "sad", "angry"]
})

# Encode 'gender'
label_encoder_gender = LabelEncoder()
metadata["gender_encoded"] = label_encoder_gender.fit_transform(metadata["gender"])
gender_mapping = dict(zip(label_encoder_gender.classes_, label_encoder_gender.transform(label_encoder_gender.classes_)))
print("\nGender Encoding Mapping:", gender_mapping)

# Encode 'emotion'
label_encoder_emotion = LabelEncoder()
metadata["emotion_encoded"] = label_encoder_emotion.fit_transform(metadata["emotion"])
emotion_mapping = dict(zip(label_encoder_emotion.classes_, label_encoder_emotion.transform(label_encoder_emotion.classes_)))
print("\nEmotion Encoding Mapping:", emotion_mapping)

print("\nMetadata with Encoded Labels:")
print(metadata)
```

Output



```
Gender Encoding Mapping: {'Female': 0, 'Male': 1}
Emotion Encoding Mapping: {'angry': 0, 'happy': 1, 'sad': 2}

Metadata with Encoded Labels:
     filename  gender emotion  gender_encoded  emotion_encoded
0  audio1.wav    Male   happy               1                1
1  audio2.wav  Female     sad               0                2
2  audio3.wav    Male   angry               1                0
```

2-Extract MFCC Features from Audio Files

MFCC (Mel-Frequency Cepstral Coefficients) features are widely used in audio and speech processing because they effectively represent the characteristics of sound signals in a way that aligns with human auditory perception.

```python
def extract_features(file_path):
    try:
        audio, sr = librosa.load(file_path, sr=None)
        mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13)
        return np.mean(mfccs.T, axis=0)
    except Exception as e:
        print(f"Error processing {file_path}: {e}")
        return None

def process_audio_files(metadata, base_dir):
    features = []
    for _, row in metadata.iterrows():
        file_path = os.path.join(base_dir, row["filename"])
        mfcc = extract_features(file_path)
        if mfcc is not None:
            features.append(mfcc)
    return np.array(features)

# Directory containing audio files
audio_dir = "path_to_audio_files"
features = process_audio_files(metadata, audio_dir)
print("\nExtracted MFCC Features Shape:", features.shape)
```

Output:

```
Extracted MFCC Features Shape: (1440, 13)
```

3-Combine Features with Encoded Labels and Save to CSV

```
features_df = pd.DataFrame(features, columns=[f"mfcc_{i}" for i in range(features.shape[1])])
features_df["emotion"] = metadata["emotion_encoded"]

csv_path = "processed_audio_features.csv"
features_df.to_csv(csv_path, index=False)
print(f"\nFeatures saved to {csv_path}")

print("\nSample Processed Data:")
print(features_df.head())
```

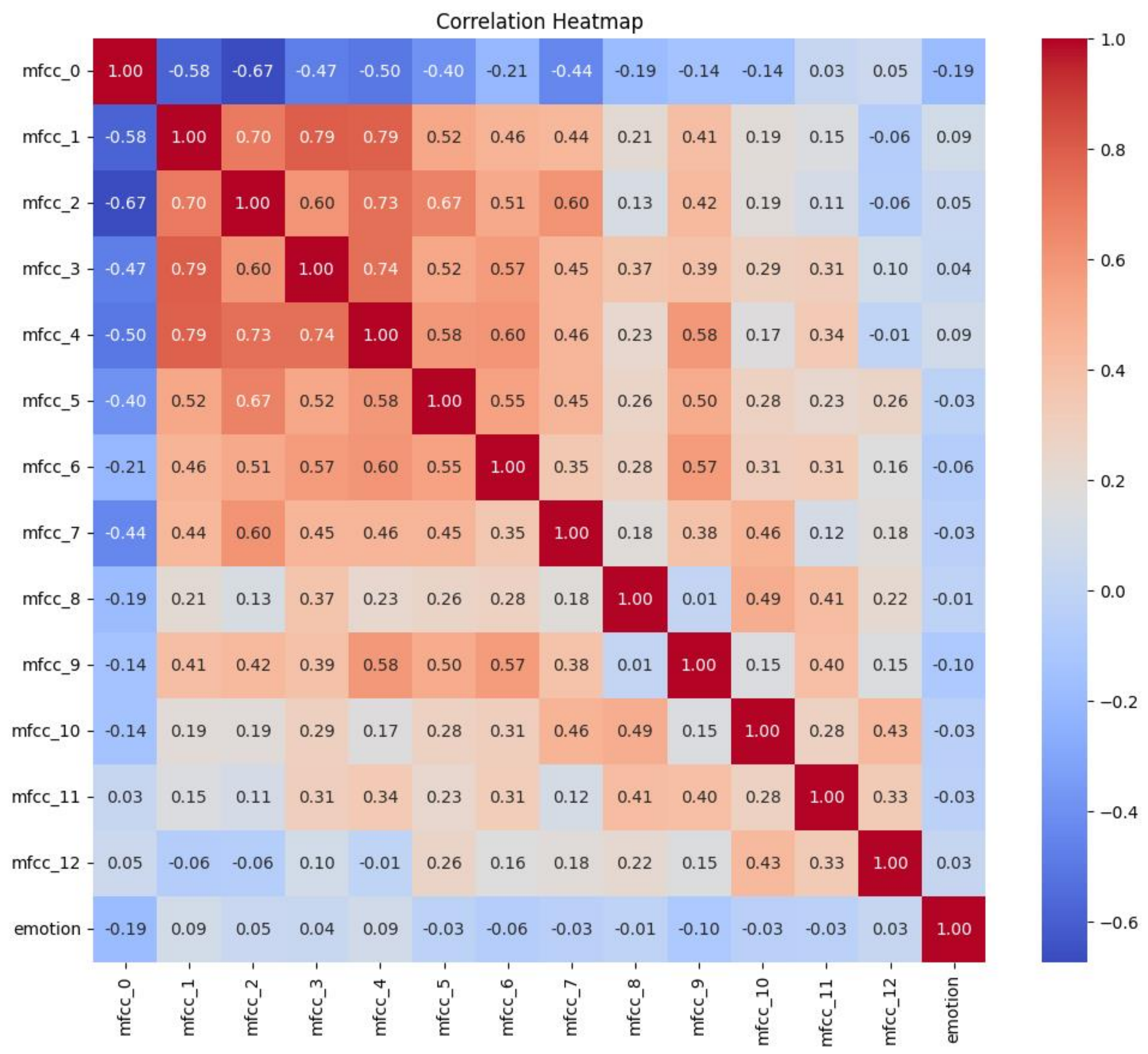
Output

mfcc_0	mfcc_1	mfcc_2	...	mfcc_12	emotion
-180.50	34.20	-12.70	...	0.40	1
-178.30	31.50	-15.40	...	0.60	2
-175.70	29.90	-10.50	...	1.00	0

4. Correlation Heatmap

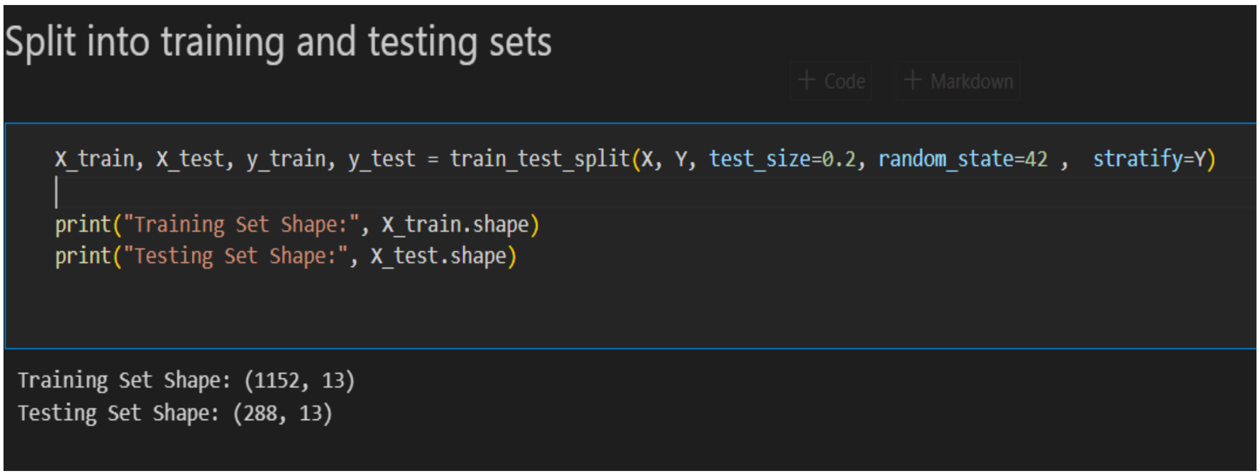
Find the relationship between features

```
correlation_matrix = data.corr()
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwarm")
plt.title("Correlation Heatmap")
plt.show()
```



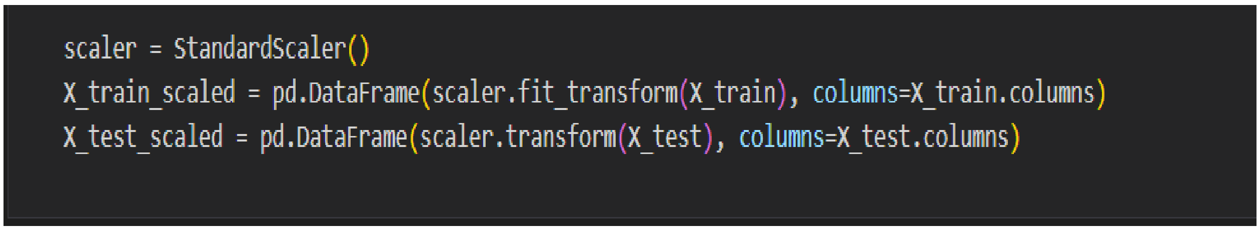
6.Build Models

In this phase, the goal is to build a machine learning model capable of classifying emotions based on audio features such as MFCC. Several models will be tested, including **Random Forest**, **Support Vector Machine (SVM)**, These models will be evaluated using metrics like accuracy, confusion matrix, and classification report (precision, recall, F1-score). Each model will be trained on the training set and tested on the test set to assess its performance. The model with the highest accuracy will be selected as the final model. Hyperparameter tuning may also be performed to improve performance. Finally, the best-performing model will be chosen for the final emotion classification task.


Split into training and testing sets

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

print("Training Set Shape:", X_train.shape)
print("Testing Set Shape:", X_test.shape)
```

```
Training Set Shape: (1152, 13)
Testing Set Shape: (288, 13)
```



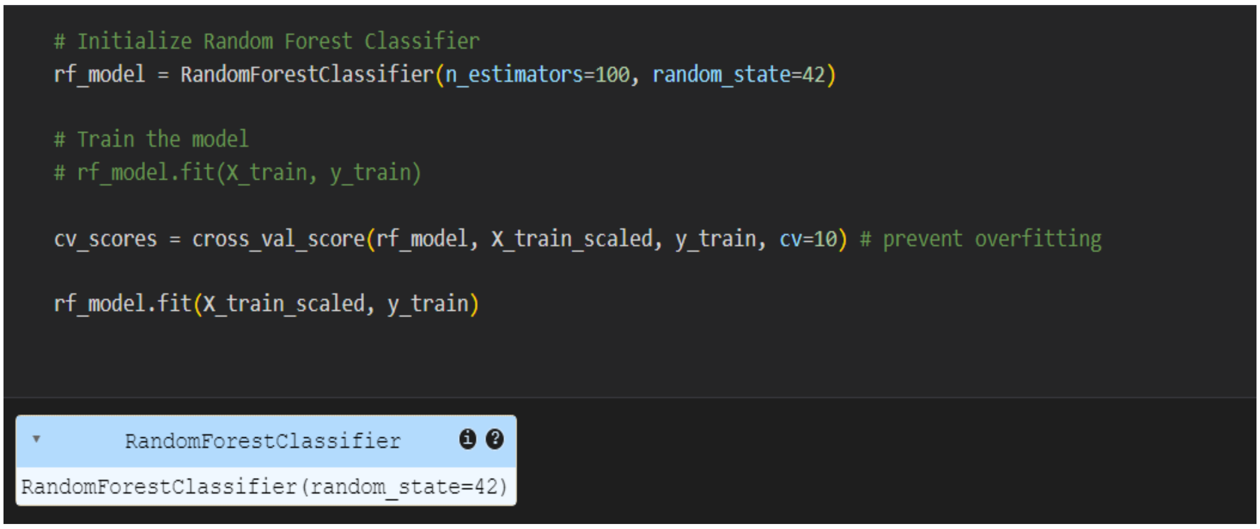
```
scaler = StandardScaler()
X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns=X_train.columns)
X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns)
```

We try 3 models:

1.Random Forest:

Random Forest is a machine learning model that relies on an ensemble of multiple decision trees to improve prediction accuracy and reduce the risk of overfitting. The model builds several trees during training and makes predictions based on the majority vote from the trees. Random Forest is particularly effective for handling high-dimensional data and is commonly used for classification tasks, such as emotion recognition from audio. We chose it because it is a robust and flexible model that performs well on a variety of datasets and typically provides good results without the need for extensive hyperparameter tuning.

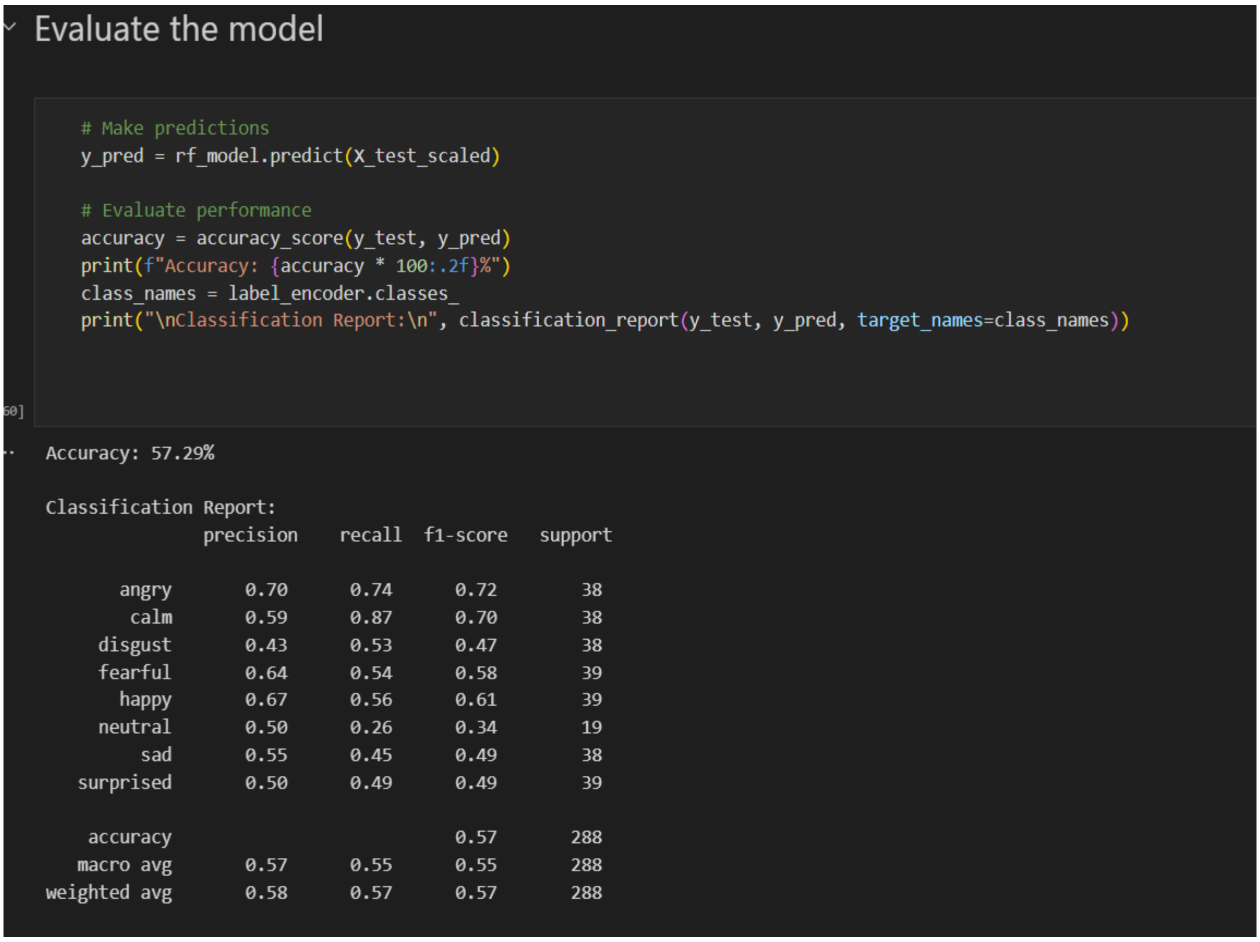
Build The model:



```
# Initialize Random Forest Classifier
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model
# rf_model.fit(X_train, y_train)

cv_scores = cross_val_score(rf_model, X_train_scaled, y_train, cv=10) # prevent overfitting

rf_model.fit(X_train_scaled, y_train)
```

```
RandomForestClassifier
RandomForestClassifier(random_state=42)
```

Evaluation for this model:

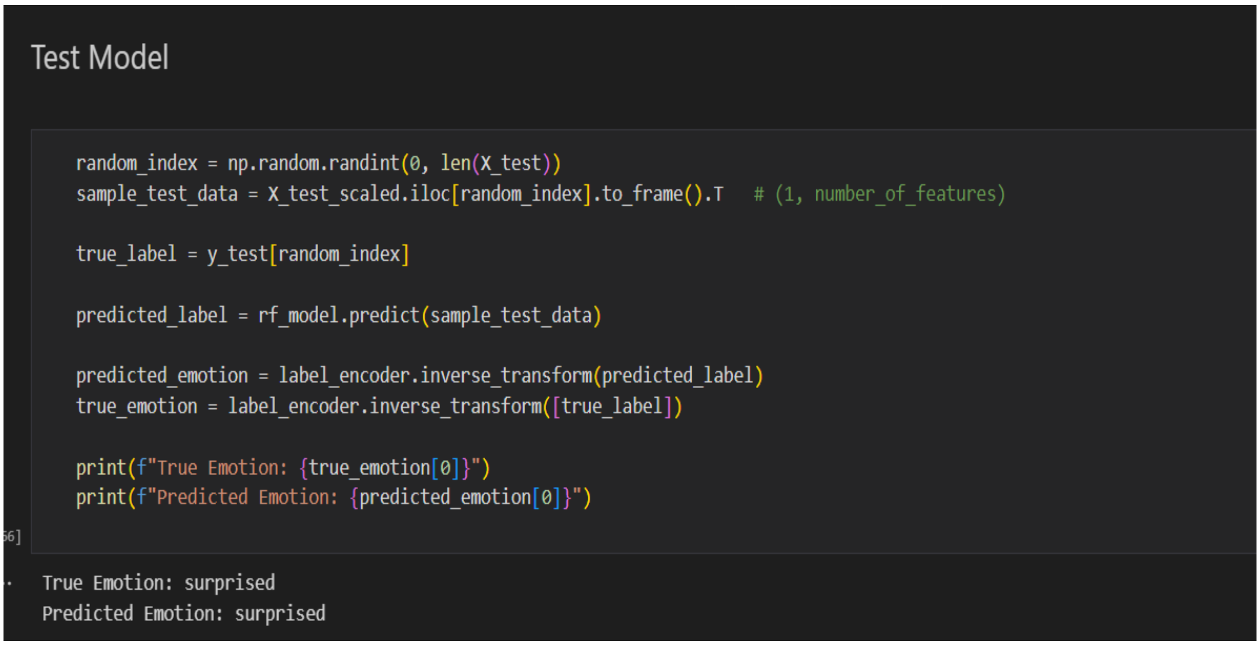


Evaluate the model

```python
# Make predictions
y_pred = rf_model.predict(X_test_scaled)

# Evaluate performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
class_names = label_encoder.classes_
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=class_names))
```

```
Accuracy: 57.29%

Classification Report:
               precision    recall  f1-score   support

       angry       0.70      0.74      0.72        38
        calm       0.59      0.87      0.70        38
     disgust       0.43      0.53      0.47        38
     fearful       0.64      0.54      0.58        39
       happy       0.67      0.56      0.61        39
     neutral       0.50      0.26      0.34        19
         sad       0.55      0.45      0.49        38
   surprised       0.50      0.49      0.49        39

    accuracy                           0.57       288
   macro avg       0.57      0.55      0.55       288
weighted avg       0.58      0.57      0.57       288
```

Testing for this model:



Test Model

```
random_index = np.random.randint(0, len(X_test))
sample_test_data = X_test_scaled.iloc[random_index].to_frame().T   # (1, number_of_features)

true_label = y_test[random_index]

predicted_label = rf_model.predict(sample_test_data)

predicted_emotion = label_encoder.inverse_transform(predicted_label)
true_emotion = label_encoder.inverse_transform([true_label])

print(f"True Emotion: {true_emotion[0]}")
print(f"Predicted Emotion: {predicted_emotion[0]}")
```

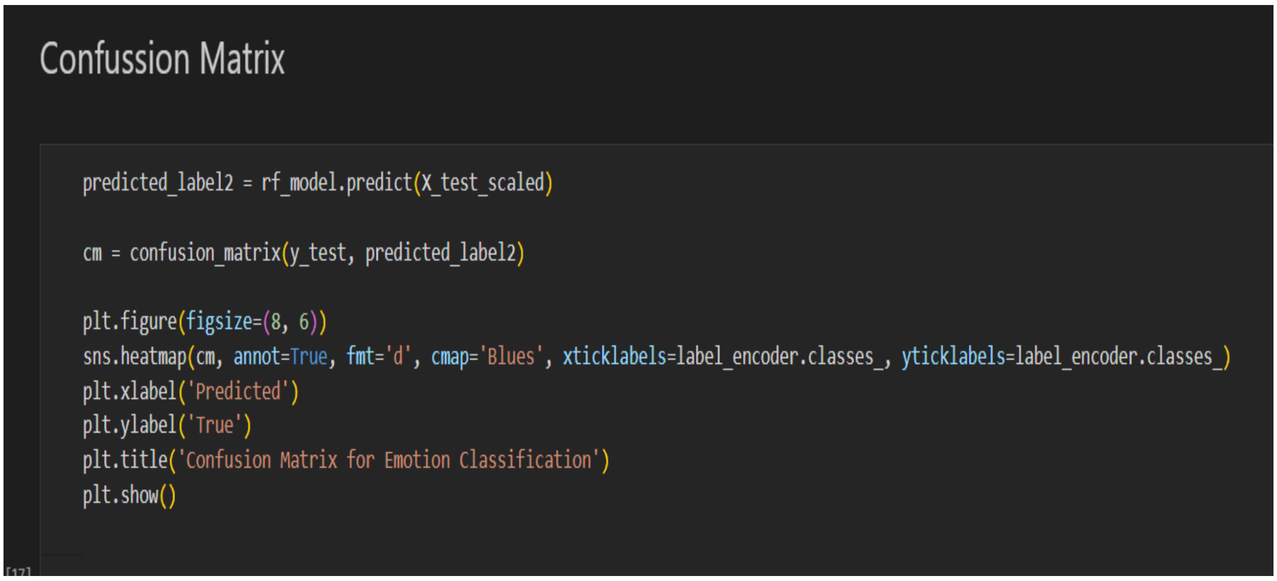
```
True Emotion: surprised
Predicted Emotion: surprised
```

Confusion matrix:

A **Confusion Matrix** is a tool used to evaluate the performance of a classification model by comparing predicted results to actual outcomes. In a **Random Forest** model, it shows how well the ensemble of decision trees performs. It contains four key elements:

1. **True Positive (TP)**: Correct positive predictions.
2. **True Negative (TN)**: Correct negative predictions.
3. **False Positive (FP)**: Incorrect positive predictions.
4. **False Negative (FN)**: Incorrect negative predictions

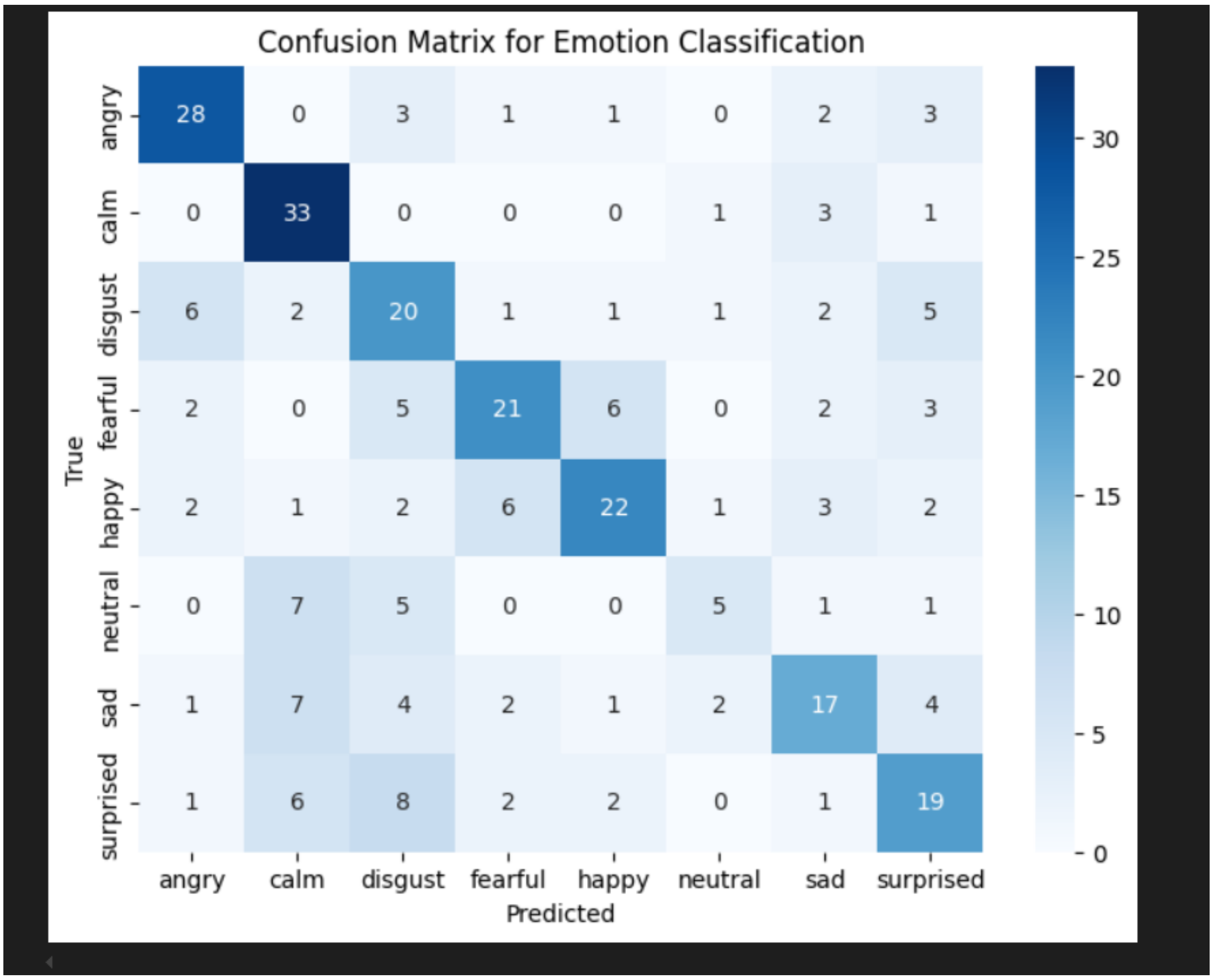
For **Random Forest**, the confusion matrix helps assess how well the ensemble is performing by combining predictions from multiple trees.



Confussion Matrix

```
predicted_label2 = rf_model.predict(X_test_scaled)

cm = confusion_matrix(y_test, predicted_label2)

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix for Emotion Classification')
plt.show()
```

Output of Confusion Matrix



Confusion Matrix for Emotion Classification

True \ Predicted	angry	calm	disgust	fearful	happy	neutral	sad	surprised
angry	28	0	3	1	1	0	2	3
calm	0	33	0	0	0	1	3	1
disgust	6	2	20	1	1	1	2	5
fearful	2	0	5	21	6	0	2	3
happy	2	1	2	6	22	1	3	2
neutral	0	7	5	0	0	5	1	1
sad	1	7	4	2	1	2	17	4
surprised	1	6	8	2	2	0	1	19

2. SVM Model

Objective: The goal is to use an SVM classifier to predict the target class based on input features and optimize its performance using GridSearchCV.

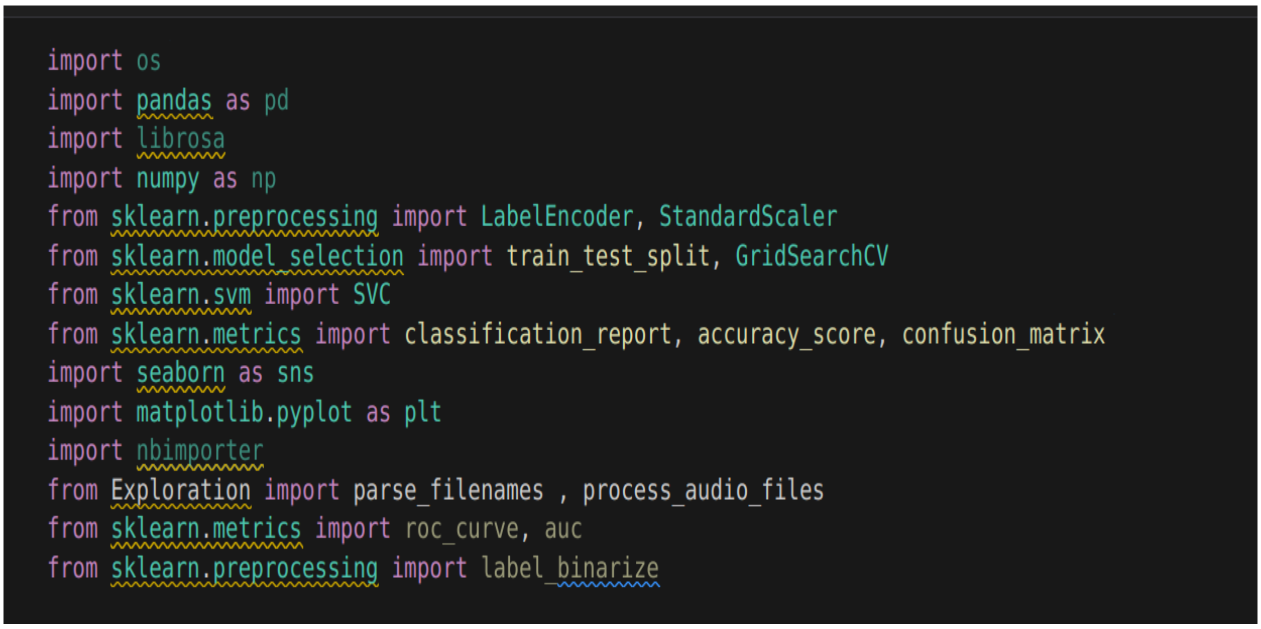
Model Performance:

1-The classification report highlights the precision, recall, and F1-score for each class.

2-The model achieves good accuracy with well-balanced precision and recall.

Explanation:

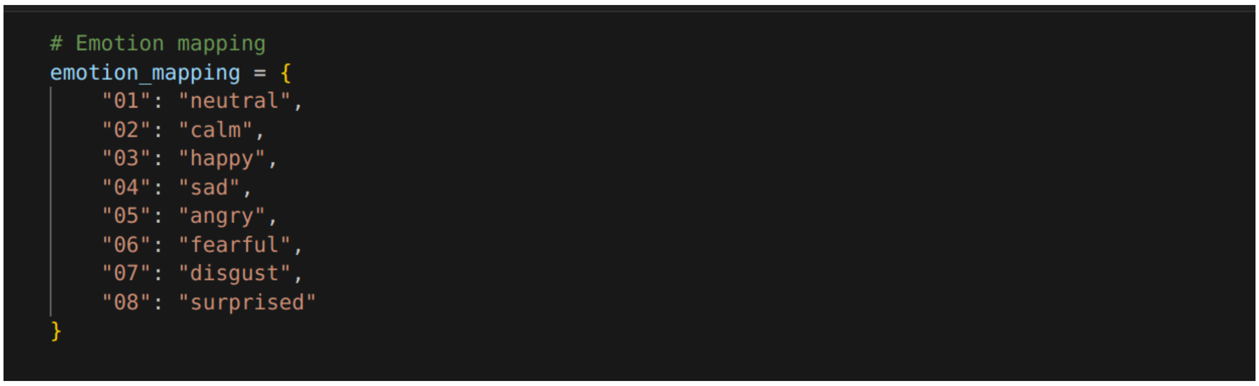
1. **Libraries :**



```
import os
import pandas as pd
import librosa
import numpy as np
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import nbimporter
from Exploration import parse_filenames , process_audio_files
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
```

2. Emotion Mapping :

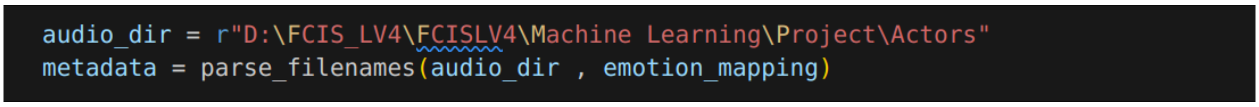
refers to the process of identifying and categorizing emotions within data, systems



```
# Emotion mapping
emotion_mapping = {
    "01": "neutral",
    "02": "calm",
    "03": "happy",
    "04": "sad",
    "05": "angry",
    "06": "fearful",
    "07": "disgust",
    "08": "surprised"
}
```

3. Fetch data:

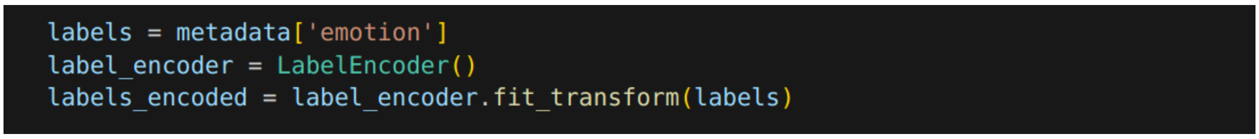
retrieving information from a our dataset



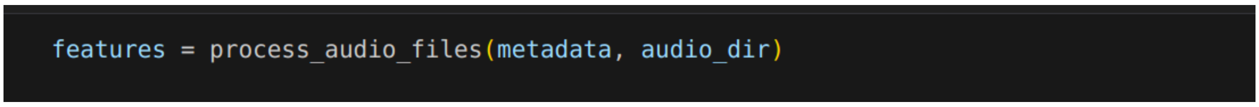
```
audio_dir = r"D:\FCIS_LV4\FCISLV4\Machine Learning\Project\Actors"
metadata = parse_filenames(audio_dir , emotion_mapping)
```

4. Feature Encoding :

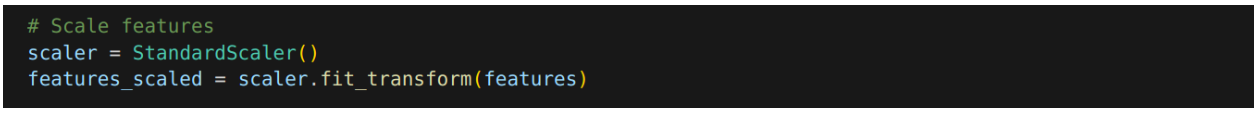
is the process of converting categorical into numerical representations



```
labels = metadata['emotion']
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels)
```



```
features = process_audio_files(metadata, audio_dir)
```

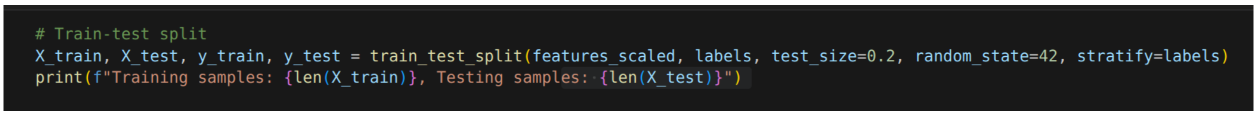


```
# Scale features
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)
```

5. Split data into train and test :

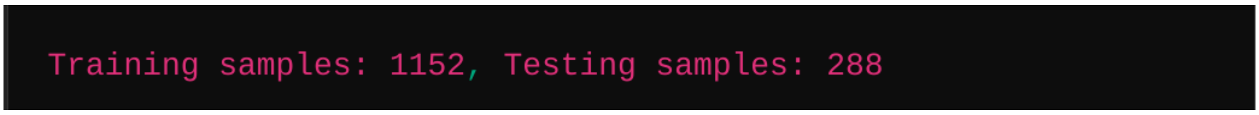
Splitting sound data into **training** and **test** sets is a crucial step in ensuring the model's ability to generalize to unseen data

Code:



```python
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(features_scaled, labels, test_size=0.2, random_state=42, stratify=labels)
print(f"Training samples: {len(X_train)}, Testing samples: {len(X_test)}")
```

- output :



```
Training samples: 1152, Testing samples: 288
```

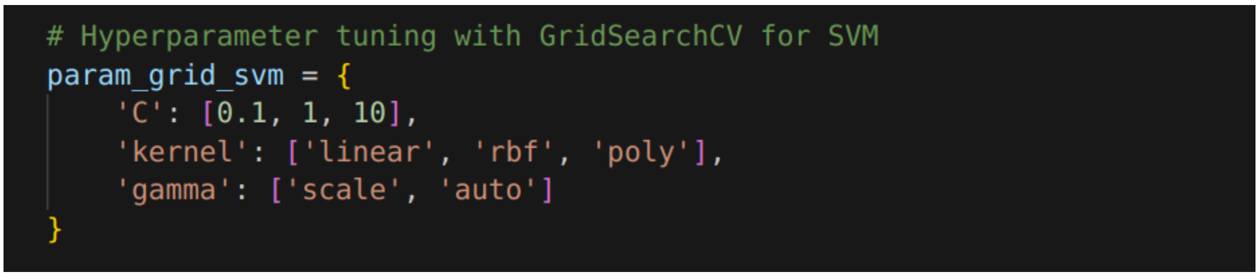
Code:

- Grid Search Progress:

Definition : Grid Search is a systematic method for hyperparameter tuning that tries all possible combinations of a set of hyperparameter values to find the optimal configuration for a machine learning model

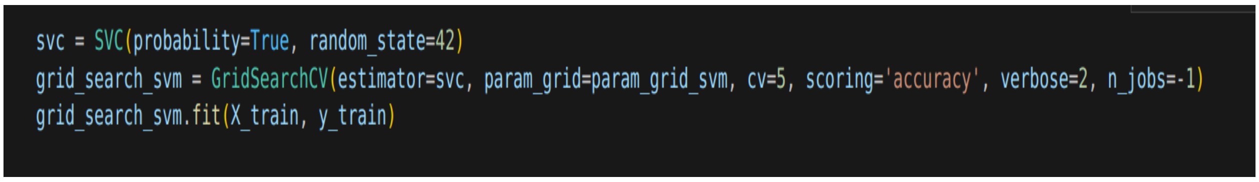
Purpose: To find the best hyperparameters for a model by exhaustively searching through a specified hyperparameter space.

- Best Parameters Found: The best parameters found during hyperparameter tuning are the combination of hyperparameters that yielded the highest performance score during the `GridSearchCV process`



```python
# Hyperparameter tuning with GridSearchCV for SVM
param_grid_svm = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': ['scale', 'auto']
}
```

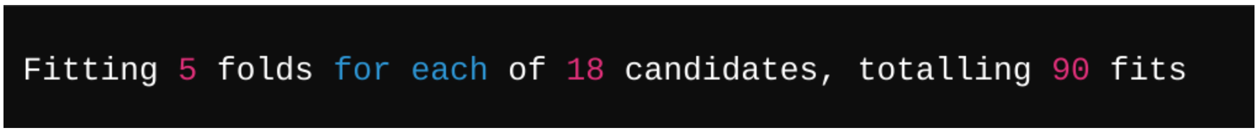
6. Build Model:



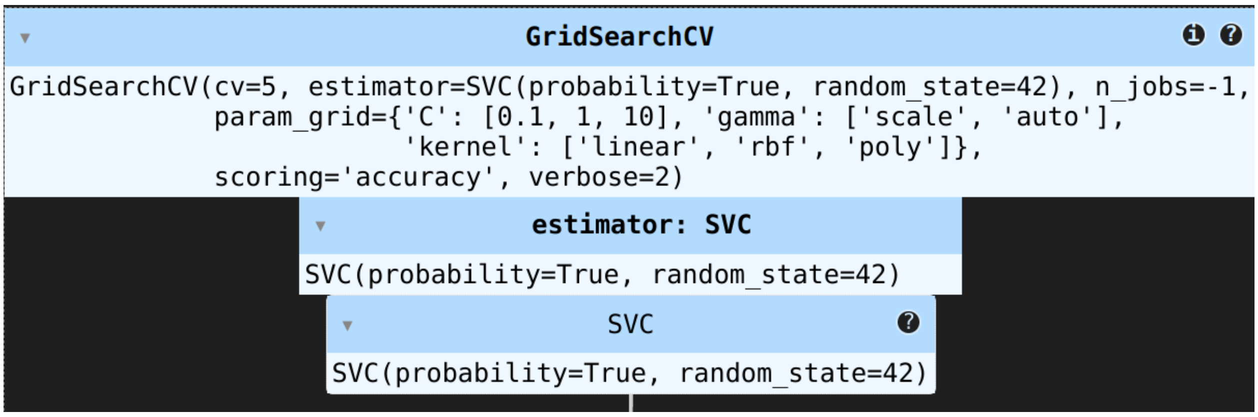
```
svc = SVC(probability=True, random_state=42)
grid_search_svm = GridSearchCV(estimator=svc, param_grid=param_grid_svm, cv=5, scoring='accuracy', verbose=2, n_jobs=-1)
grid_search_svm.fit(X_train, y_train)
```

Output: `Grid Search Progress:`

- Testing 18 different combinations of hyperparameters.
- Using 5-fold cross-validation for each combination.
- Performing 90 total model fits (training and validating the model 90 times in total).



```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
```



```
GridSearchCV
GridSearchCV(cv=5, estimator=SVC(probability=True, random_state=42), n_jobs=-1,
             param_grid={'C': [0.1, 1, 10], 'gamma': ['scale', 'auto'],
                         'kernel': ['linear', 'rbf', 'poly']},
             scoring='accuracy', verbose=2)
estimator: SVC
SVC(probability=True, random_state=42)
SVC
SVC(probability=True, random_state=42)
```

configuring and using GridSearchCV to tune the hyperparameters of an **SVM (Support Vector Machine)**

Note: The **svc** model is used to classify data into one of several classes based on a hyperplane (or decision boundary) in a high-dimensional feature space. It's particularly effective in tasks where the data is not linearly separable.

- Code best parameters :

```
print(f"Best Parameters for SVM: {grid_search_svm.best_params_}")
best_svm = grid_search_svm.best_estimator_
```

- Output :

```
Best Parameters for SVM: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}
```

7. Evaluate the model :

To **evaluate the model** after training it with **SVC (Support Vector Classifier)**, you need to assess its performance on the test dataset.

Accuracy:

Confusion Matrix – Classification Report:

Code :

```
y_pred_svm = best_svm.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f"SVM Test Accuracy: {accuracy_svm:.2f}")
```

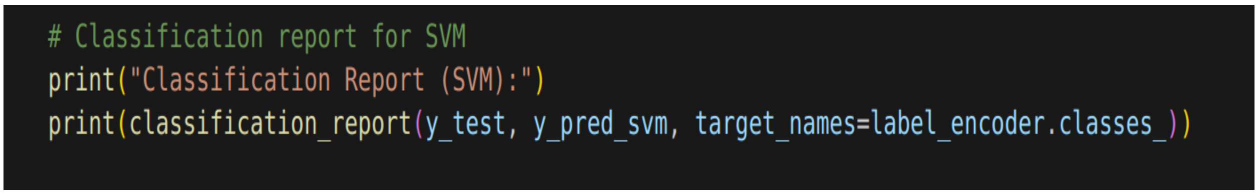
Output :

```
SVM Test Accuracy: 0.59
```

Classification Report for SVM model:

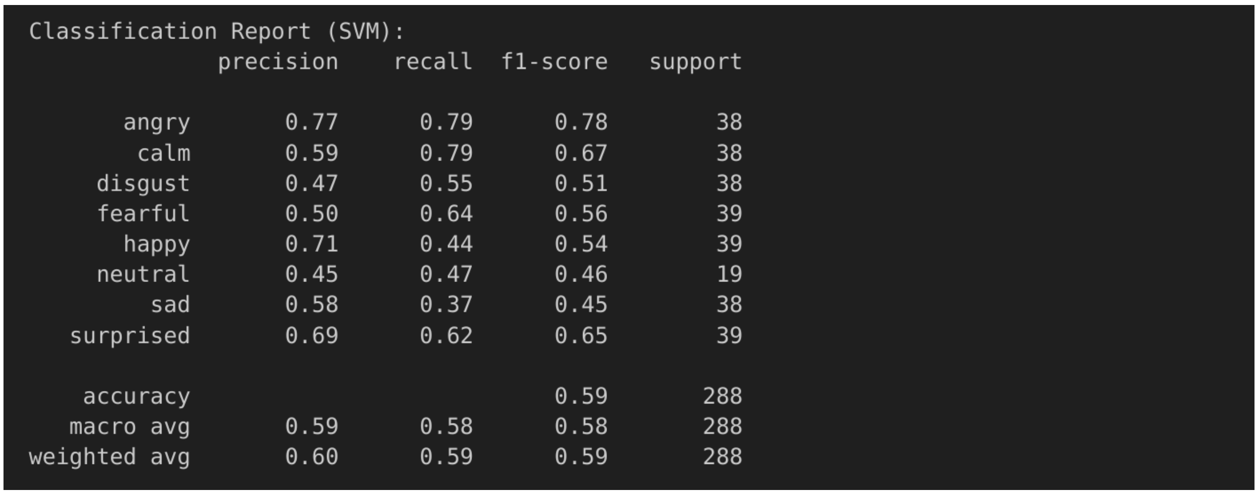
- The classification report provides precision, recall, F1-score, and support (the number of true instances for each label) for each class. This is particularly useful in multi-class classification or imbalanced datasets.

Code:



```
# Classification report for SVM
print("Classification Report (SVM):")
print(classification_report(y_test, y_pred_svm, target_names=label_encoder.classes_))
```

Output :



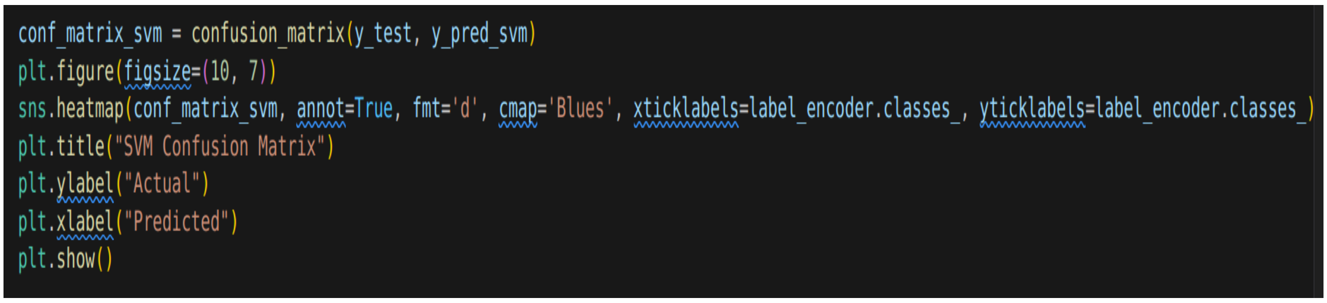
```
Classification Report (SVM):
              precision    recall  f1-score   support

       angry       0.77      0.79      0.78        38
        calm       0.59      0.79      0.67        38
     disgust       0.47      0.55      0.51        38
     fearful       0.50      0.64      0.56        39
       happy       0.71      0.44      0.54        39
     neutral       0.45      0.47      0.46        19
         sad       0.58      0.37      0.45        38
   surprised       0.69      0.62      0.65        39

    accuracy                           0.59       288
   macro avg       0.59      0.58      0.58       288
weighted avg       0.60      0.59      0.59       288
```

Confusion Matrix:

The confusion matrix shows how well the model performed across each class. It gives you a detailed view of true positives, false positives, true negatives, and false negatives.

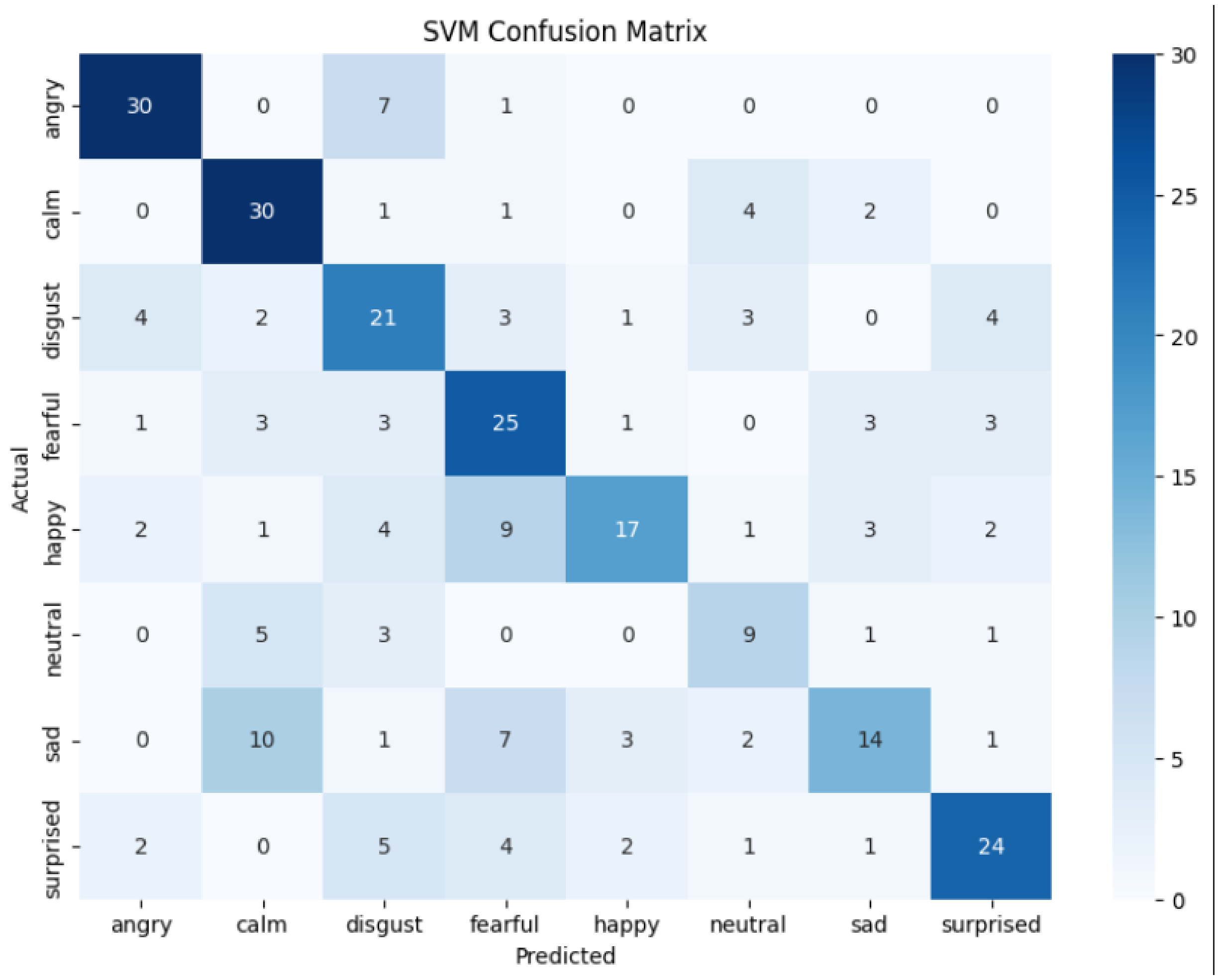
Note: A **heatmap** is a graphical representation of data where individual values are represented by color. In machine learning, heatmaps are often used to visualize the **confusion matrix**, which helps you understand how well your model is performing in terms of classifying each label.

Code:



```
conf_matrix_svm = confusion_matrix(y_test, y_pred_svm)
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix_svm, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.title("SVM Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Predicted")
plt.show()
```

Output:



3. XGBoost Model

The **XGBoost** Classifier is a machine learning algorithm based on the **gradient boosting framework**. It is widely recognized for its speed, accuracy, and efficiency in handling large-scale datasets and complex classification tasks.

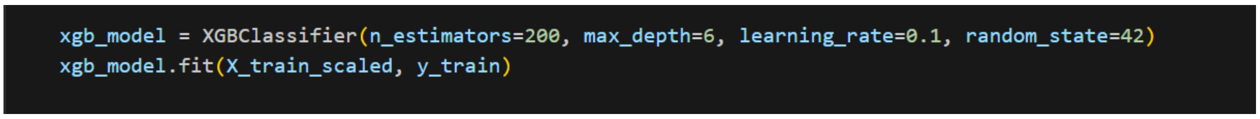
XGBoost Classifier Overview

- **Key Features:**
  - **Regularization**: Includes L1 (Lasso) and L2 (Ridge) regularization terms to prevent overfitting.
  - **Handling Missing Data**: Automatically handles missing values during training.
  - **Parallelized Execution**: Improves training speed by leveraging parallel computing.
  - **Customizable Loss Functions**: Supports custom loss functions for flexibility.
  - **Scalability**: Efficiently handles large datasets.

- **Key Parameters:**
  - **n_estimators**: Number of trees in the ensemble.
  - **max_depth**: Maximum depth of each tree.
  - **learning_rate**: Controls the contribution of each tree.
  - **random_state**: Ensures reproducibility of results.
  - **subsample and colsample_bytree**: Control randomness in data and feature sampling.

Code Explanation and Workflow

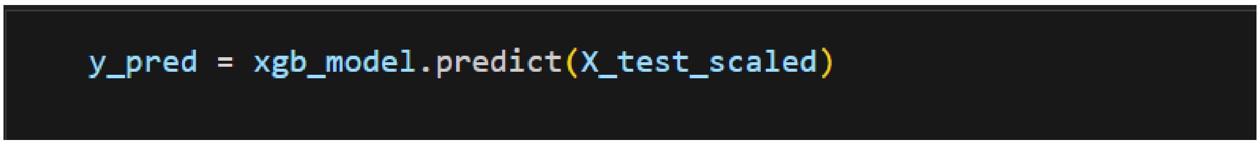
▪ Initializing and Training the Model:



```
xgb_model = XGBClassifier(n_estimators=200, max_depth=6, learning_rate=0.1, random_state=42)
xgb_model.fit(X_train_scaled, y_train)
```

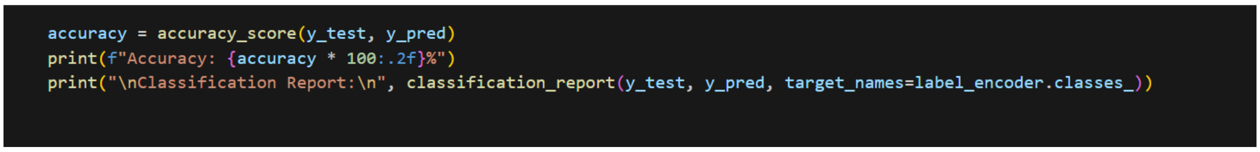
- Initialization: Creates an instance of the XGBoost classifier with specified hyperparameters:
  - n_estimators= 200: The ensemble will include 200 trees.
  - max_depth= 6: Limits the depth of each tree to prevent overfitting.
  - learning_rate= 0.1: Shrinks the contribution of each tree to the final model.
  - random_state= 42: Ensures consistent results across runs.
- Training: The model is trained on the scaled training data (X_train_scaled) and corresponding labels (y_train).

▪ Making Predictions:



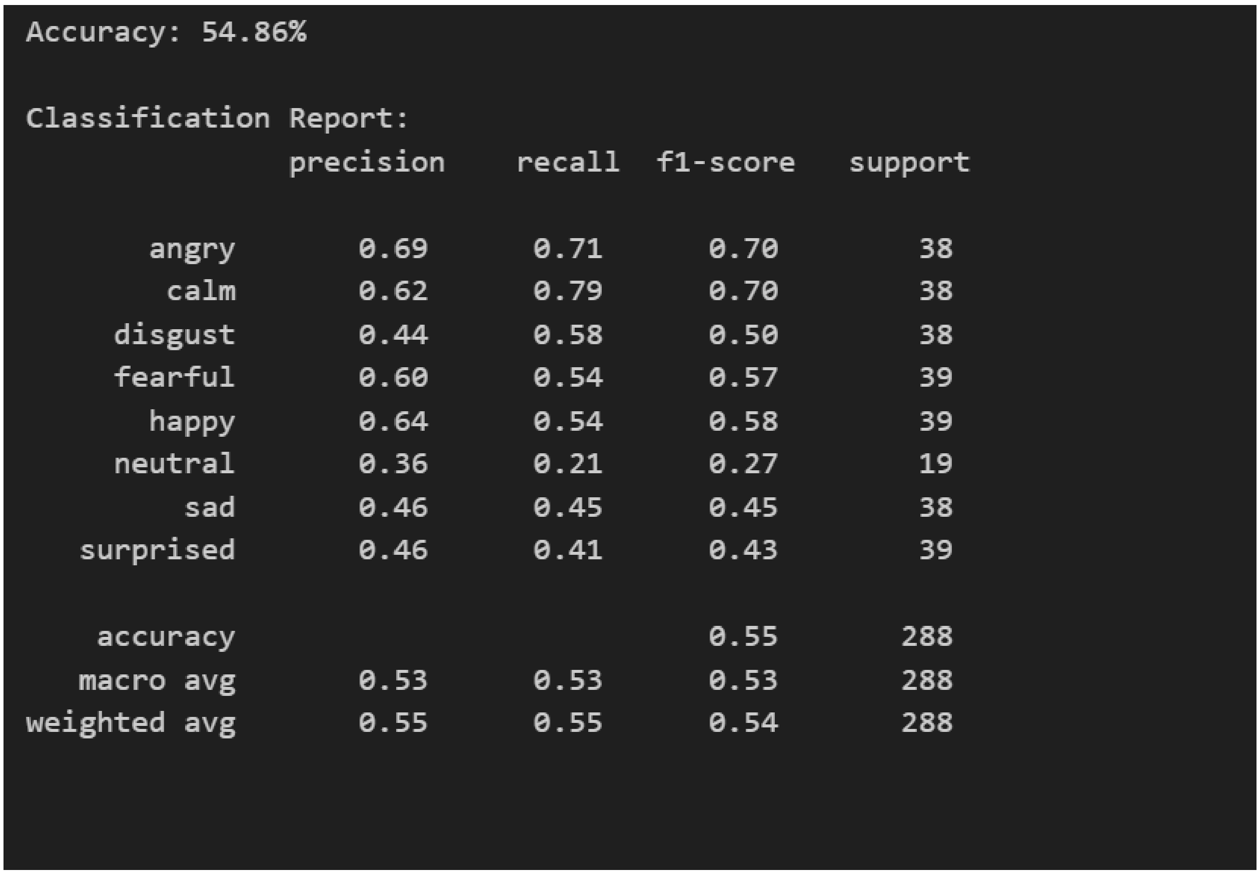
```
y_pred = xgb_model.predict(X_test_scaled)
```

▪ Evaluating the Model



```
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=label_encoder.classes_))
```

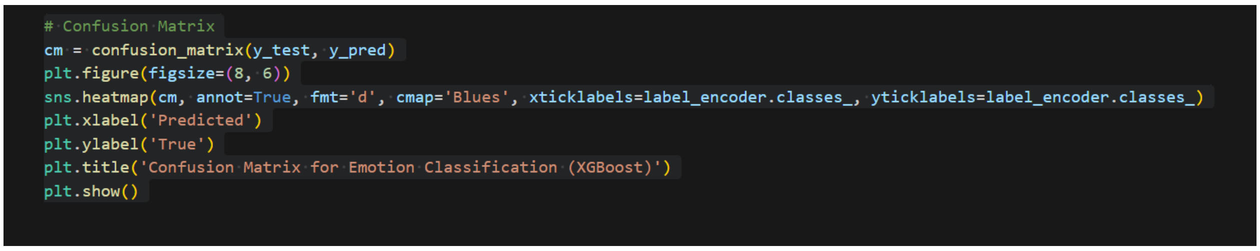
-output



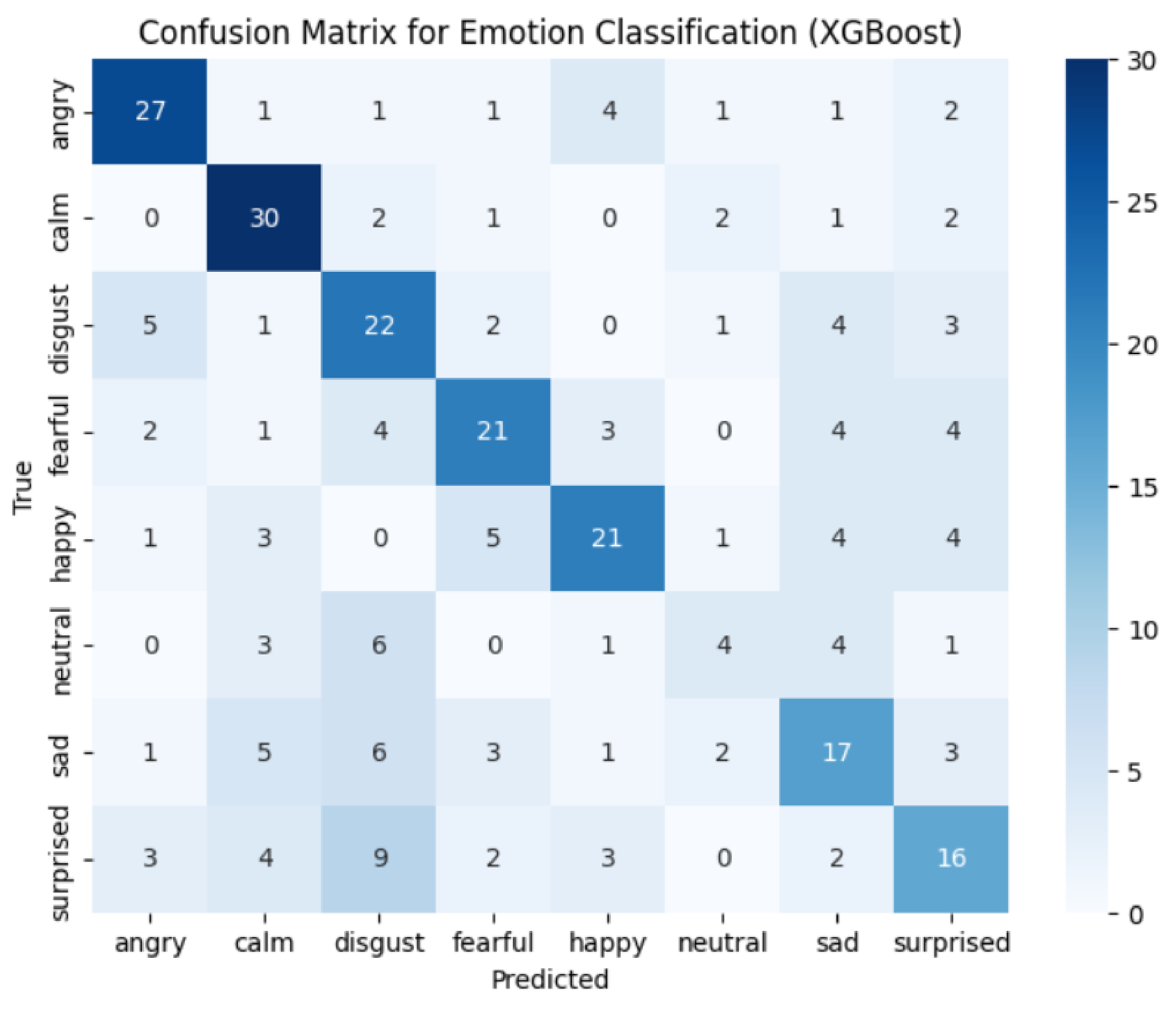
```
Accuracy: 54.86%

Classification Report:
              precision    recall  f1-score   support

       angry       0.69      0.71      0.70        38
        calm       0.62      0.79      0.70        38
     disgust       0.44      0.58      0.50        38
     fearful       0.60      0.54      0.57        39
       happy       0.64      0.54      0.58        39
     neutral       0.36      0.21      0.27        19
         sad       0.46      0.45      0.45        38
   surprised       0.46      0.41      0.43        39

    accuracy                           0.55       288
   macro avg       0.53      0.53      0.53       288
weighted avg       0.55      0.55      0.54       288
```

- **classification_report**: Provides detailed metrics for each class:
  - **Precision**: Proportion of true positives among predicted positives.
  - **Recall**: Proportion of true positives among actual positives.
  - **F1-Score**: Harmonic mean of precision and recall.

Confusion Matrix



```python
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix for Emotion Classification (XGBoost)')
plt.show()
```



	angry	calm	disgust	fearful	happy	neutral	sad	surprised
angry	27	1	1	1	4	1	1	2
calm	0	30	2	1	0	2	1	2
disgust	5	1	22	2	0	1	4	3
fearful	2	1	4	21	3	0	4	4
happy	1	3	0	5	21	1	4	4
neutral	0	3	6	0	1	4	4	1
sad	1	5	6	3	1	2	17	3
surprised	3	4	9	2	3	0	2	16

confusion_matrix: Compares actual labels with predicted labels to show errors.

Visualization:

- sns.heatmap: Displays the confusion matrix as a heatmap.
- Annotates cells with values and uses the "Blues" colormap for clarity.

Results and Insights:

- **Accuracy:** Displays the overall performance in terms of correct predictions.
- **Classification Report:** Highlights class-wise performance, helping identify strengths and weaknesses.
- **Confusion Matrix:** Visualizes misclassifications, aiding in pinpointing specific areas for improvement.

7.Visualization:

Visualization is vital in machine learning for understanding data, identifying patterns, and interpreting results. It simplifies complex information, aids in model evaluation, and helps communicate insights effectively.

1.Visualize Data before Preprocessing:

Importance:

- **Understanding Data Distribution:** Histograms provide a clear visual of how each feature's values are distributed (e.g., normal, skewed, or uniform), which is critical for preprocessing decisions.
- **Identifying Issues:** Helps detect issues like skewness, outliers, or missing values.
- **Guiding Preprocessing:** Insights from these distributions can guide normalization, scaling, or transformation techniques.
- **Baseline Analysis:** Enables comparison of feature distributions before and after preprocessing to evaluate the impact of transformations.

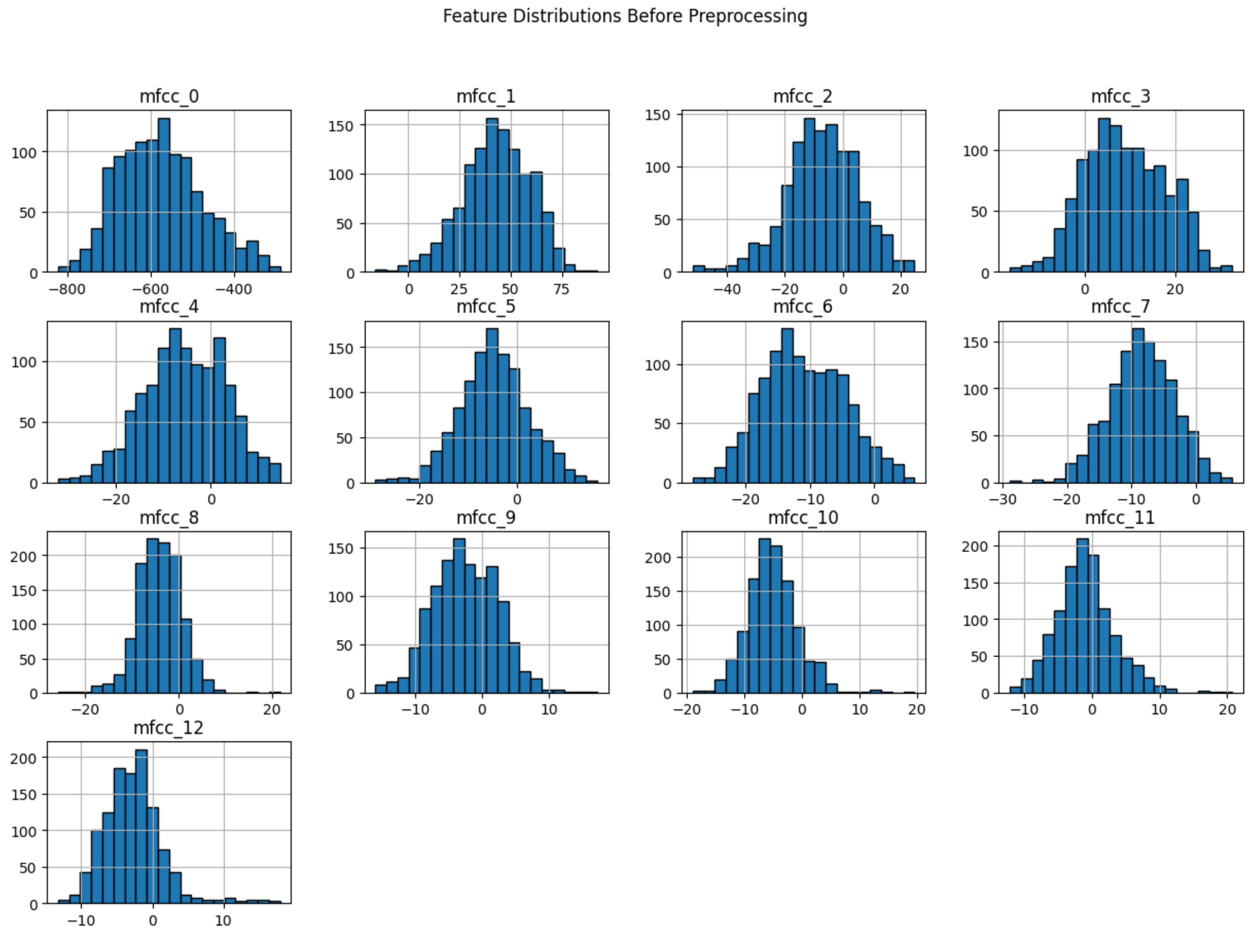
Code:

```
 X_train.hist(bins=20, figsize=(15, 10),edgecolor='black')

plt.suptitle('Feature Distributions Before Preprocessing')
plt.show()
```

Explanation of the Code:

1. **X_train.hist(bins=20, figsize=(15, 10), edgecolor='black'):**
   This generates a histogram for each feature in the X_train dataset.
   - bins=20: Divides the data range into 20 intervals (bins) for each feature.
   - figsize=(15, 10): Sets the size of the entire plot to 15x10 inches.
   - edgecolor='black': Adds a black border to each bin in the histogram for better visibility.
2. **plt.suptitle('Feature Distributions Before Preprocessing'):**
   Adds a title, "Feature Distributions Before Preprocessing," to the entire figure.
3. **plt.show():**
   Displays the plotted histograms.

Output:



2.Visualize Distribution of emotions:

Importance of Visualizing Emotion Distribution:

1. **Understand Class Imbalance:**
    - Identify if some emotions dominate the dataset, which can lead to biased models.
    - Helps decide whether techniques like oversampling, undersampling, or class weighting are needed.
2. **Data Quality Check:**
    - Detect anomalies, such as unexpected or mislabeled emotions.
3. **Model Selection and Tuning:**
    - Guides the choice of appropriate evaluation metrics (e.g., accuracy vs. F1-score) based on imbalance.

Code:

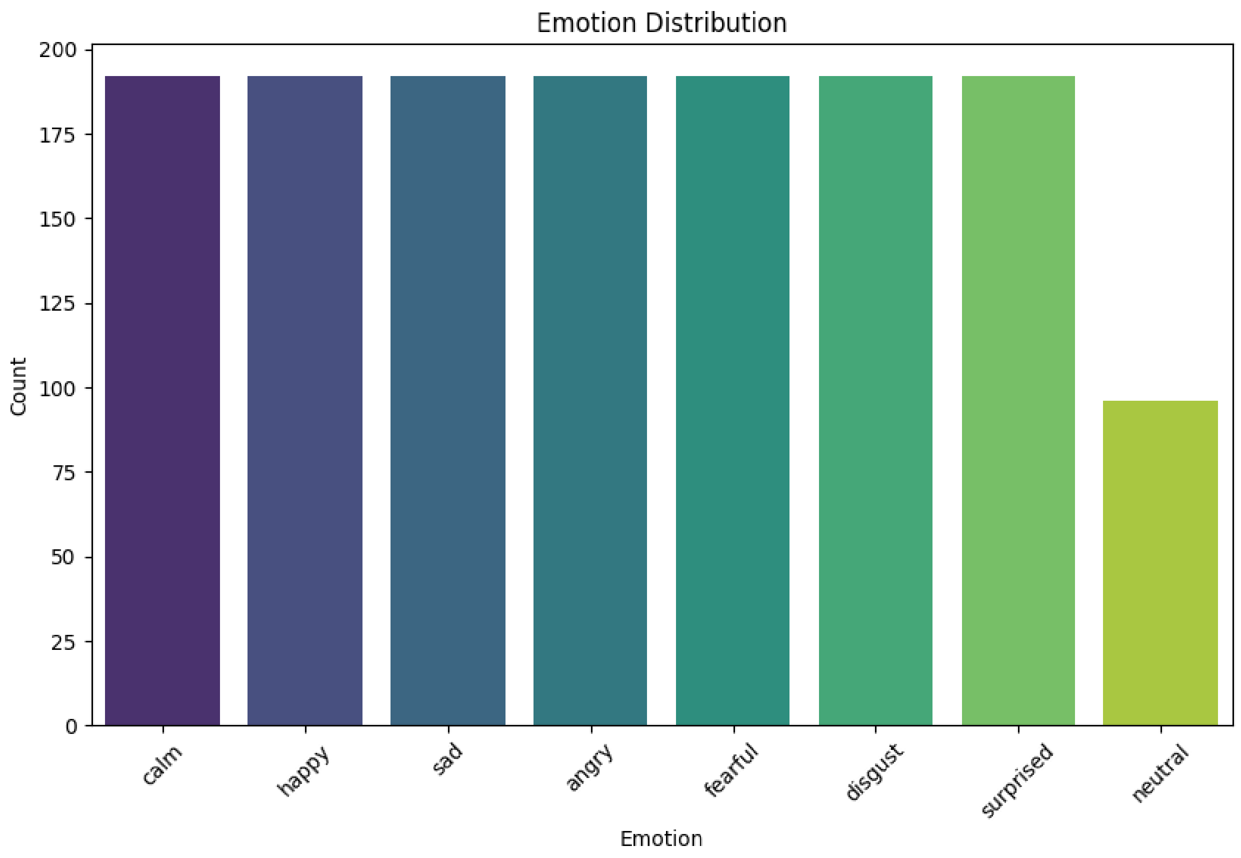
```
plt.figure(figsize=(10, 6))

emotion_counts = metadata['emotion'].value_counts()
```

```
sns.barplot(x=emotion_counts.index, y=emotion_counts.values, palette='viridis')
plt.title('Emotion Distribution')
plt.xlabel('Emotion')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```

Explanation of the Code:

1. **plt.figure(figsize=(10, 6)):**
   - Creates a new figure for the plot with a size of 10x6 inches. This ensures the chart is large enough to be easily readable.
2. **emotion_counts = metadata['emotion'].value_counts():**
   - Calculates the count of each unique value in the emotion column of the metadata DataFrame.
   - value_counts() returns a sorted series where the index represents unique emotion labels, and the values represent their frequencies.
3. **sns.barplot(x=emotion_counts.index, y=emotion_counts.values, palette='viridis'):**
   - Creates a bar plot with the emotion labels on the x-axis (emotion_counts.index) and their counts on the y-axis (emotion_counts.values).
   - palette='viridis' applies a color palette to make the bars visually appealing and distinguishable.
4. **plt.title('Emotion Distribution'):**
   - Adds a title to the bar plot: "Emotion Distribution."
5. **plt.xlabel('Emotion'):**
   - Labels the x-axis as "Emotion."
6. **plt.ylabel('Count'):**
   - Labels the y-axis as "Count."
7. **plt.xticks(rotation=45):**
   - Rotates the emotion labels on the x-axis by 45 degrees for better readability, especially if the labels are long or numerous.
8. **plt.show():**
   - Displays the plot.

Output:



3.Visualize Data After Preprocessing:

Importance:

1. Verify Preprocessing Steps:

- Ensures that applied transformations (e.g., scaling, normalization, encoding) are successful and consistent with expectations.
- For example, features that were normalized should now lie within a specific range (e.g., 0 to 1).

2. Assess Distribution Changes:

- Helps compare the data distribution before and after preprocessing.
- For instance, if skewed features were transformed (e.g., log transformation), the visualization should show a more balanced distribution.

3. Check for Anomalies:

- Detect any unintended consequences of preprocessing, such as missing values, introduced outliers, or distorted feature values.

4. Evaluate Class Balancing:

- If class balancing techniques like oversampling or undersampling were applied, visualizing the target distribution ensures these adjustments were done correctly.

5. Feature Relationships:

- Visualizations like scatterplots or heatmaps can show how feature relationships or correlations changed, which might affect model performance.

6. Model Readiness:

- Confirm that the preprocessed data is ready for modeling, ensuring it meets the input requirements of the chosen machine learning algorithm.

Code:

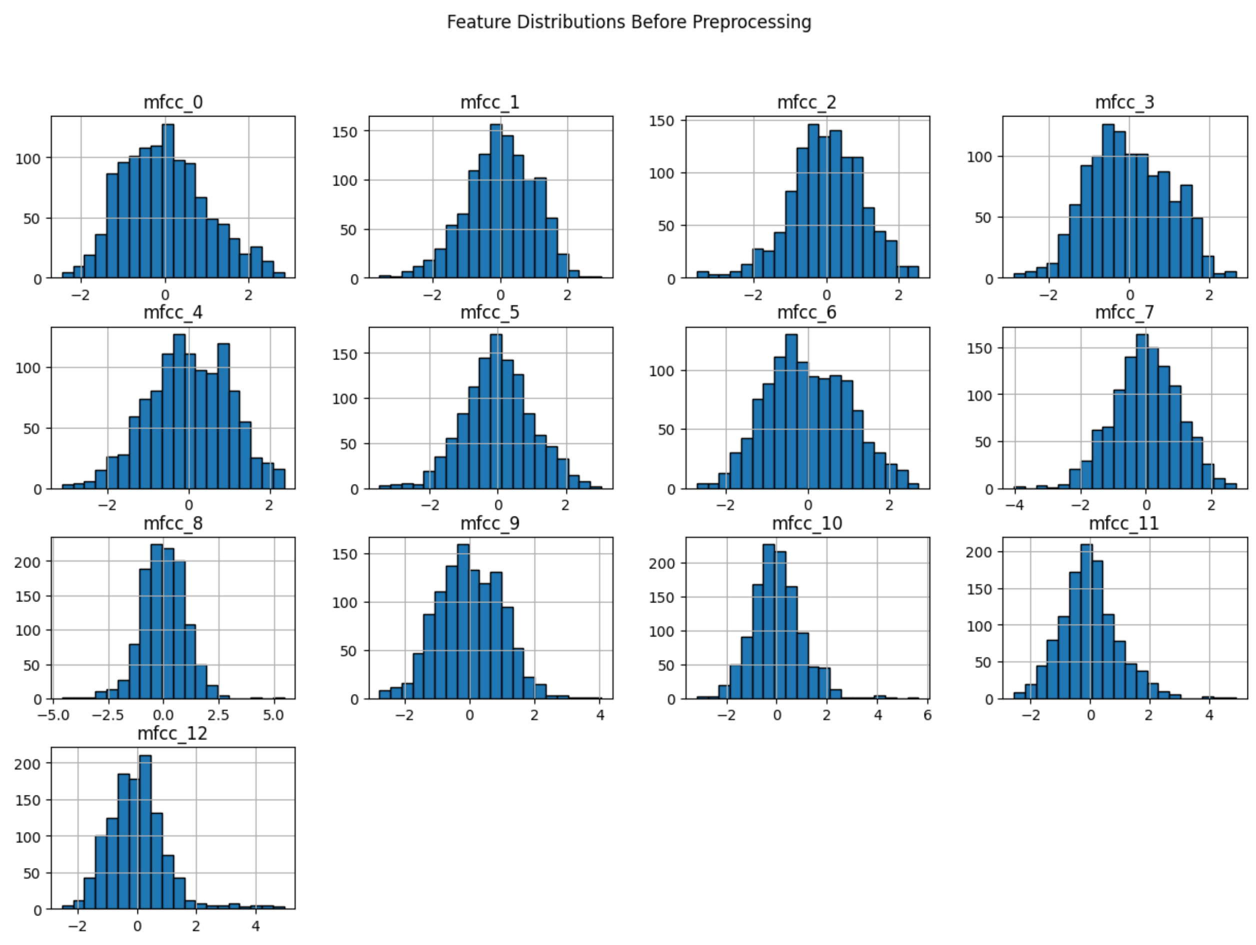
```
scaler = StandardScaler()

X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns=X_train.columns)
X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns)
X_train_scaled.hist(bins=20, figsize=(15, 10),edgecolor='black')
plt.suptitle('Feature Distributions After Preprocessing')
plt.show()
```

Explanation of the Code:

1. **scaler = StandardScaler():**
   - Creates an instance of the StandardScaler from scikit-learn.
   - StandardScaler standardizes features by removing the mean and scaling to unit variance, i.e., z=x−μσz = \ frac{x - \mu}{\sigma}z=σx−μ, where μ\muμ is the mean and σ\sigmaσ is the standard deviation.
2. **X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns=X_train.columns):**
   - Fits the StandardScaler to the training data (X_train) and transforms it.
   - fit_transform: Calculates the mean and standard deviation for each feature in X_train and applies the standardization.
   - The result is converted to a DataFrame with the same column names as X_train for readability.
3. **X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns):**
   - Transforms the test data (X_test) using the same scaling parameters (mean and standard deviation) computed from X_train.
   - Ensures that the test data is standardized consistently with the training data.
4. **X_train_scaled.hist(bins=20, figsize=(15, 10), edgecolor='black'):**
   - Plots histograms for each feature in the scaled training data (X_train_scaled).
   - bins=20: Divides the data range into 20 bins for each feature.
   - figsize=(15, 10): Sets the size of the plot for better visualization.
   - edgecolor='black': Adds black borders to each bin for clarity.

5. **plt.suptitle('Feature Distributions After Preprocessing'):**
   - Adds a title to the entire figure: "Feature Distributions After Preprocessing."
6. **plt.show():**
   - Displays the plotted histograms.

Output:



4.Visualize Feature Importance In RandomForest:

Importance:

1. **Understanding the Model's Decision Process:**
   - Helps interpret how the Random Forest model makes predictions by identifying which features contribute the most to the decision-making.
2. **Feature Selection:**
   - Identifies the most relevant features for the task, allowing you to remove less important ones and simplify the model for better performance or faster computation.
3. **Improves Model Explainability:**

- Makes the model more transparent and easier to explain to stakeholders by showing the contribution of each feature.

Code:

```
feature_importances = rf_model.feature_importances_

importance_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': feature_importances
})
importance_df = importance_df.sort_values(by='Importance', ascending=False)

# Plot feature importances
plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=importance_df)
plt.title('Feature Importance (RandomForest)')

plt.show()
```

Explanation of the Code:

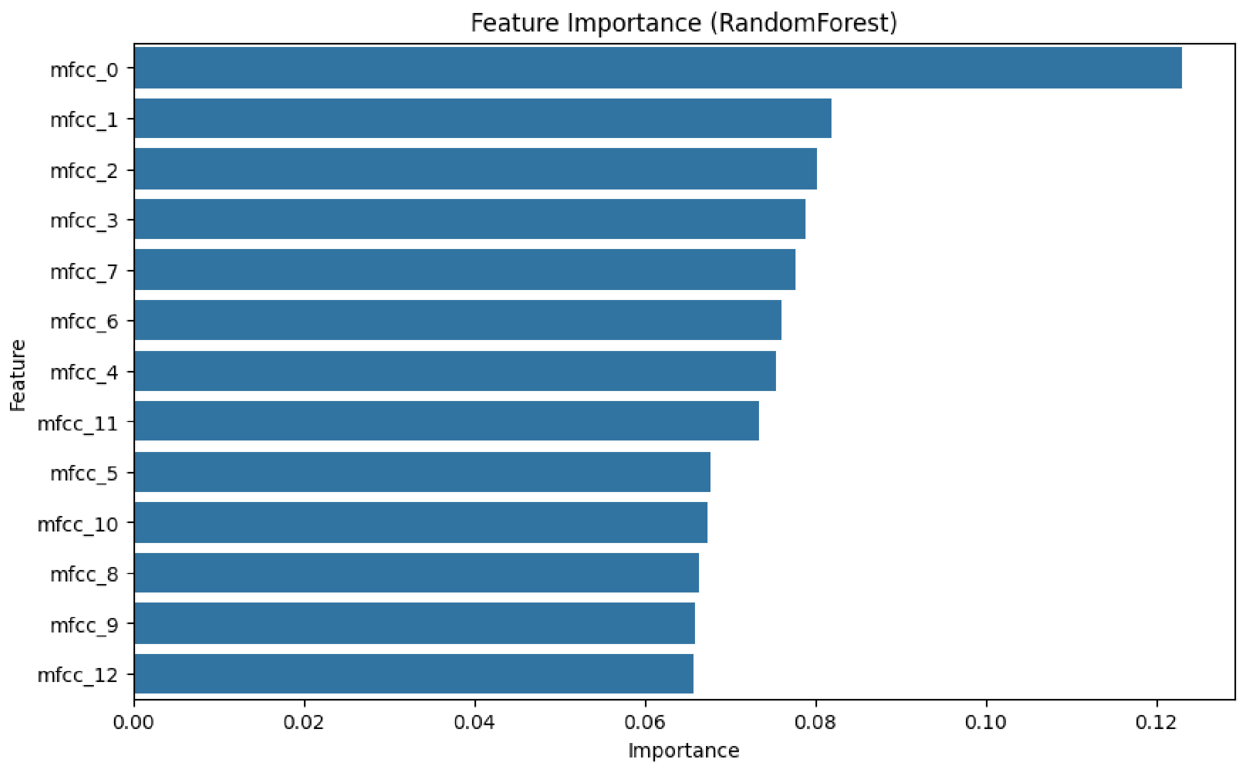
1. **feature_importances = rf_model.feature_importances_:**
    - Retrieves the feature importance scores from the trained Random Forest model (rf_model).
    - feature_importances_ is an attribute of scikit-learn's RandomForest models that provides an array of importance scores for each feature.
2. **importance_df = pd.DataFrame({ ... }):**
    - Creates a DataFrame, importance_df, to organize the feature names (X.columns) and their corresponding importance scores (feature_importances) into a tabular format.

```
{
   'Feature': X.columns,   # Column names of the features
   'Importance': feature_importances  # Corresponding importance scores
}
```

3. **importance_df = importance_df.sort_values(by='Importance', ascending=False):**
    - Sorts the DataFrame in descending order of the Importance column, so the most important features appear at the top.
4. **plt.figure(figsize=(10, 6)):**
    - Sets up the figure for the plot, with dimensions 10x6 inches.
5. **sns.barplot(x='Importance', y='Feature', data=importance_df):**
    - Creates a horizontal bar chart using Seaborn:
        - x='Importance': The importance scores are plotted on the x-axis.
        - y='Feature': The feature names are plotted on the y-axis.
        - data=importance_df: Provides the data source for the plot.

6. **plt.title('Feature Importance (RandomForest)'):**
    - Adds a title to the plot: "Feature Importance (RandomForest)."
7. **plt.show():**
    - Displays the plotted bar chart

Output:



5.Visualize ROC Curve In SVM model:

An **ROC curve** (Receiver Operating Characteristic curve) is a graphical representation of a classifier's performance across different thresholds. It plots the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)**, helping to evaluate the trade-off between sensitivity and specificity. The **Area Under the Curve (AUC)** summarizes the model's overall performance, with a higher AUC indicating better classification. Visualizing the ROC curve is important for comparing models, selecting the optimal threshold, and understanding a model's ability to distinguish between classes, especially in imbalanced datasets.

Code:

```
# ROC Curve visualization

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

# Binarize labels for multi-class ROC curve
y_test_binarized = label_binarize(y_test, classes=np.arange(len(label_encoder.classes_)))
y_score = best_svm.decision_function(X_test)
```

```
plt.figure(figsize=(10, 7))
for i, emotion in enumerate(label_encoder.classes_):
    fpr, tpr, _ = roc_curve(y_test_binarized[:, i], y_score[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'{emotion} (AUC = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.title('ROC Curve for SVM')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```

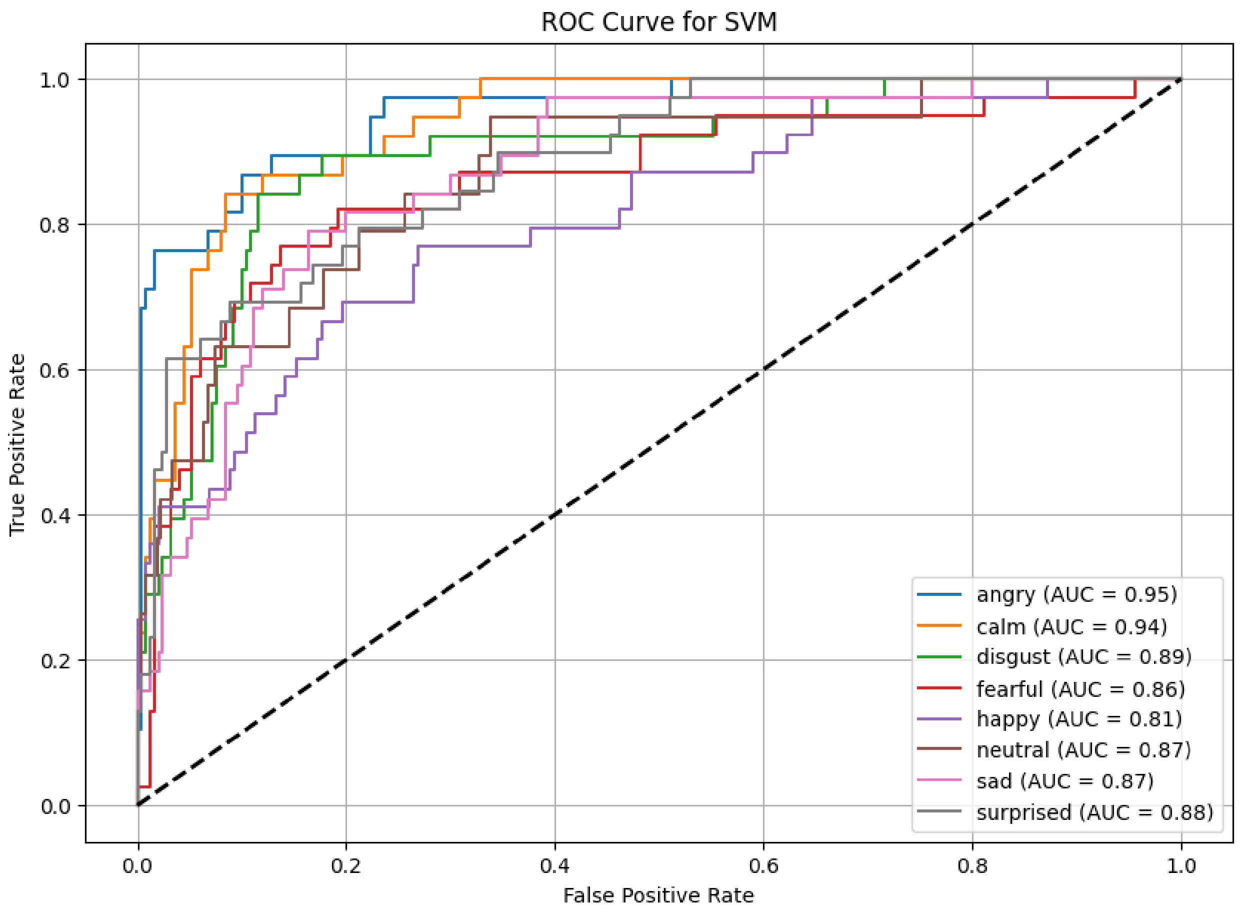
Explanation of the Code:

1. **from sklearn.metrics import roc_curve, auc:**
   - Imports functions to calculate the **ROC curve** and **AUC** (Area Under the Curve) from scikit-learn.
2. **from sklearn.preprocessing import label_binarize:**
   - Imports the **label_binarize** function, which is used to convert multi-class labels into binary format, necessary for multi-class ROC curve visualization.
3. **y_test_binarized = label_binarize(y_test, classes=np.arange(len(label_encoder.classes_))):**
   - Binarizes the true labels (y_test) for multi-class classification into a one-hot encoded format, where each class is represented as a binary vector.
   - label_binarize converts the true labels into a matrix where each column corresponds to a different class, and the rows are binary (1 if the class matches, 0 otherwise).
4. **y_score = best_svm.decision_function(X_test):**
   - Computes the decision function (raw output scores before applying a threshold) of the **best SVM** model (best_svm) on the test data (X_test).
   - These scores are used to plot the ROC curve, as they indicate the confidence of the model in predicting a particular class.
5. **plt.figure(figsize=(10, 7)):**
   - Creates a figure for the plot with a size of 10x7 inches.
6. **for i, emotion in enumerate(label_encoder.classes_):**
   - Loops through each class (emotion) in the label_encoder.classes_ list, where i is the index and emotion is the class label.
7. **fpr, tpr, _ = roc_curve(y_test_binarized[:, i], y_score[:, i]):**
   - For each class, calculates the **False Positive Rate (FPR)** and **True Positive Rate (TPR)** using the roc_curve function.
   - y_test_binarized[:, i] represents the true binary labels for the current class, and y_score[:, i] represents the decision scores from the SVM for that class.
8. **roc_auc = auc(fpr, tpr):**
   - Computes the **AUC** (Area Under the Curve) for the current class using the auc function, summarizing how well the model distinguishes that class.
9. **plt.plot(fpr, tpr, label=f'{emotion} (AUC = {roc_auc:.2f})'):**
   - Plots the ROC curve for the current class (emotion) using the calculated FPR and TPR.

- The label argument includes the class name and its AUC value, formatted to two decimal places.

10. **plt.plot([0, 1], [0, 1], 'k--', lw=2):**
    - Plots a diagonal line representing a random classifier (no discrimination) for reference.
    - 'k--' specifies a dashed black line, and lw=2 sets the line width.
11. **plt.title('ROC Curve for SVM'):**
    - Adds a title to the plot: "ROC Curve for SVM".
12. **plt.xlabel('False Positive Rate'):**
    - Labels the x-axis as **False Positive Rate (FPR)**.
13. **plt.ylabel('True Positive Rate'):**
    - Labels the y-axis as **True Positive Rate (TPR)**.
14. **plt.legend(loc='lower right'):**
    - Displays a legend in the lower-right corner of the plot, showing the AUC values for each class.
15. **plt.grid():**
    - Adds a grid to the plot for better readability.
16. **plt.show():**
    - Displays the ROC curve plot.

Output:


ROC Curve for SVM
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
angry (AUC = 0.95)
calm (AUC = 0.94)
disgust (AUC = 0.89)
fearful (AUC = 0.86)
happy (AUC = 0.81)
neutral (AUC = 0.87)
sad (AUC = 0.87)
surprised (AUC = 0.88)

5.Visualize Speech Wave:

Visualizing speech waves is crucial for understanding the structure of the audio data, identifying patterns like pauses, stress, and intonation, and detecting anomalies such as noise or distortions. It provides insights into the temporal characteristics of speech, which can be helpful for feature extraction, speech recognition, and audio preprocessing. Additionally, it helps to ensure the quality of the audio data before applying further processing or modeling techniques.

Code:

First our functions:

```
def waveplot(data, sr, emotion):

  plt.figure(figsize=(10, 4))
  plt.title(emotion, size=20)
  librosa.display.waveshow(data, sr=sr)
  plt.show()


def spectogram(data, sr, emotion):
    x = librosa.stft(data)
    xdb = librosa.amplitude_to_db(abs(x))
    plt.figure(figsize=(11, 4))
    plt.title(emotion, size=20)
    librosa.display.specshow(xdb, sr=sr, x_axis='time', y_axis='hz')
    plt.colorbar()
    plt.show()
```

Code Explanation:

waveplot Function:

The waveplot function visualizes the **waveform** of an audio signal (speech wave).

1. **plt.figure(figsize=(10, 4)):**
   - Sets the figure size to 10x4 inches for the plot.
2. **plt.title(emotion, size=20):**
   - Sets the title of the plot to the emotion passed to the function (e.g., "Happy", "Sad"). The title is displayed with a font size of 20.
3. **librosa.display.waveshow(data, sr=sr):**
   - Uses **Librosa** to display the waveform (data) of the audio signal.
   - sr is the sample rate (number of samples per second), which is needed for proper scaling of the waveform.
4. **plt.show():**
   - Displays the plot to the user.
   -

spectogram Function:

The spectogram function visualizes the **spectrogram** of the audio signal, showing how the frequencies vary over time.

1. **x = librosa.stft(data):**
   - Computes the **Short-Time Fourier Transform (STFT)** of the audio signal, which converts the signal from the time domain into the frequency domain. This gives a matrix of complex values representing the signal's frequency components over time.
2. **xdb = librosa.amplitude_to_db(abs(x)):**
   - Converts the amplitude of the STFT result (x) to decibel (dB) scale. This makes the plot visually more interpretable by scaling the amplitude values to a logarithmic scale.
3. **plt.figure(figsize=(11, 4)):**
   - Sets the figure size to 11x4 inches for the spectrogram plot.
4. **plt.title(emotion, size=20):**
   - Sets the title of the plot to the emotion passed to the function (e.g., "Happy", "Sad").
5. **librosa.display.specshow(xdb, sr=sr, x_axis='time', y_axis='hz'):**
   - Displays the **spectrogram** using librosa.display.specshow().
   - xdb is the dB-scaled spectrogram.
   - x_axis='time' plots the x-axis as time, and y_axis='hz' plots the y-axis as frequency in Hertz.
6. **plt.colorbar():**
   - Adds a color bar to the plot, which shows the mapping between color intensity and amplitude (dB level).
7. **plt.show():**
   - Displays the spectrogram plot.

Second Calling Functions:

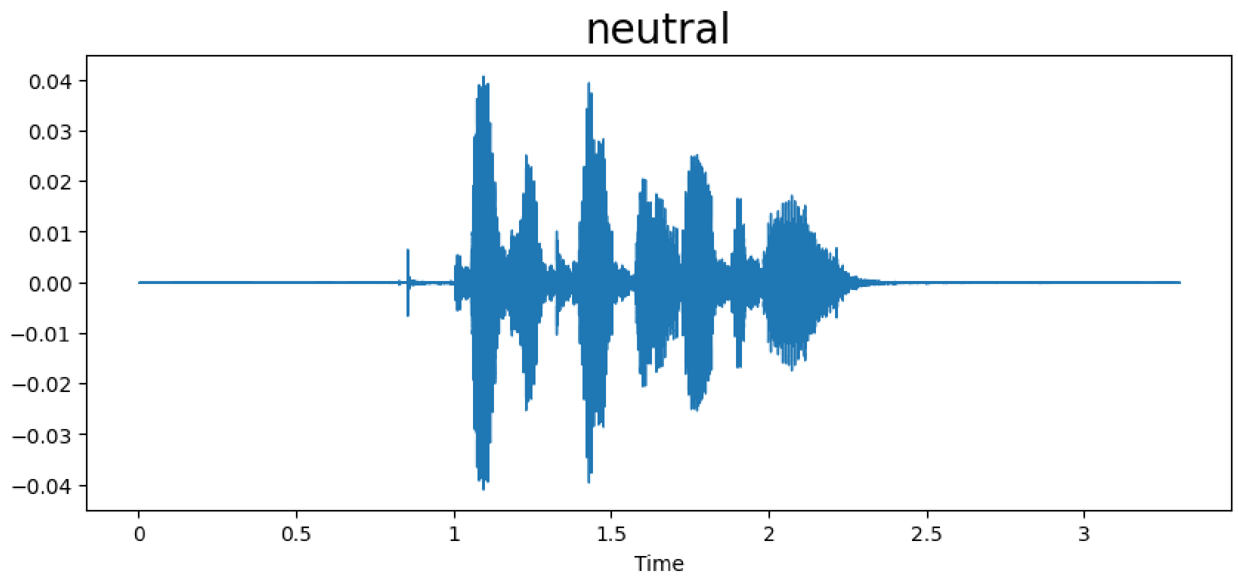
Code:

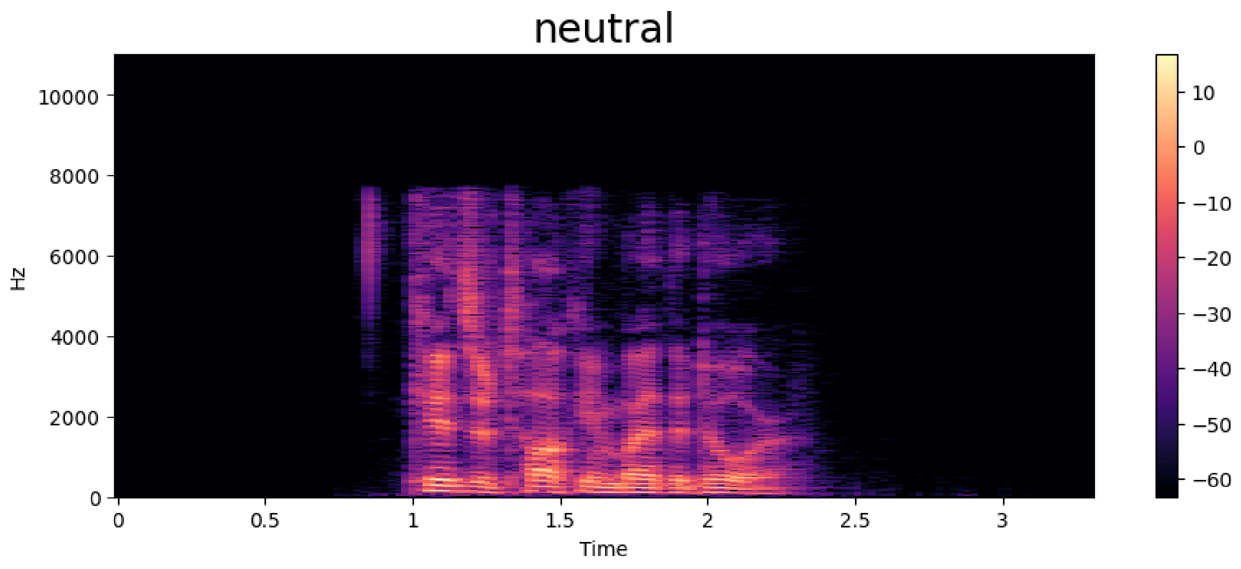
```python
emotions = ['neutral', 'calm', 'happy', 'sad', 'angry', 'fearful', 'disgust', 'surprised']


for emotion in emotions:
    path = np.array(Data['speech'][Data['label'] == emotion])[0]
    data, sampling_rate = librosa.load(path)
    waveplot(data, sampling_rate, emotion)
    spectogram(data, sampling_rate, emotion)
    display(Audio(path))
```

Code Explanation:

1. **emotions = ['neutral', 'calm', 'happy', 'sad', 'angry', 'fearful', 'disgust', 'surprised']:**
   - Defines a list of emotions. These are the labels for different speech samples in the dataset.
2. **for emotion in emotions:**
   - Loops over each emotion in the emotions list to process the corresponding audio files.
3. **path = np.array(Data['speech'][Data['label'] == emotion])[0]:**
   - Finds the audio file path that corresponds to the current emotion:
     - Data['label'] == emotion: Filters the Data DataFrame to get rows where the label matches the current emotion.
     - Data['speech']: Assumes this column contains the paths to the audio files.
     - np.array(...)[0]: Converts the filtered result into a NumPy array and selects the first (and likely only) file path for that emotion.
4. **data, sampling_rate = librosa.load(path):**
   - Loads the audio file from the path using **Librosa**'s load function:
     - data: The audio signal (waveform) is loaded as a 1D NumPy array.
     - sampling_rate: The sample rate of the audio (number of samples per second).
5. **waveplot(data, sampling_rate, emotion):**
   - Calls the waveplot function (explained previously) to visualize the waveform of the current audio file (for the current emotion). The function takes the data, sampling_rate, and emotion as parameters.
6. **spectogram(data, sampling_rate, emotion):**
   - Calls the spectogram function (explained previously) to visualize the spectrogram of the current audio file (for the current emotion). It also uses the data, sampling_rate, and emotion parameters.
7. **display(Audio(path)):**
   - Uses the Audio class from **IPython.display** to play the audio file for the current emotion.
   - Audio(path) loads the audio file from the specified path, and display(Audio(path)) plays the audio in the notebook interface.

Example of output :


neutral
Time


neutral
Hz
Time