Sentiment Analysis from Audio Recordings

(Code Report)

Muhammad Qasim[1]
(SP – 2022 – BSCS – 048)

Awais Ali[1]
(SP – 2022 – BSCS – 080)

Sohaib Ahmad[1]
(SP – 2022 – BSCS – 060)

Hammad Ahmad[1]
(SP – 2022 – BSCS – 063)

[1]Department of Computer Science, Lahore Garrison University, Pakistan

Abstract

Sentiment analysis, also known as opinion mining, is a subfield of natural language processing and machine learning that focuses on identifying and interpreting the emotional tone expressed in various forms of data, including text, audio, and video. In this project, we explore sentiment analysis from audio recordings, aiming to classify the emotions conveyed in speech. Audio sentiment analysis is particularly valuable for applications such as customer feedback analysis, monitoring social media sentiments, and gauging audience reactions in real time. Using the CREMA-D dataset, which comprises 7,442 audio clips labelled with different emotions, we implemented a robust methodology to tackle this challenge. The pipeline includes noise removal, feature extraction through Mel Frequency Cepstral Coefficients (MFCCs), dimensionality reduction via Principal Component Analysis (PCA), and classification using machine learning algorithms. Classifiers such as K-Nearest Neighbours (KNN), Logistic Regression, Support Vector Machines (SVM), Naive Bayes, and Neural Networks were evaluated using metrics like accuracy, precision, recall, and F1-score. The SVM classifier outperformed others with an accuracy of 45.39%, followed by Logistic Regression at 43.69%. Although KNN, Naïve Bayes, and Neural Networks achieved moderate results, they provided insights into challenges such as data complexity and emotion variability. These findings highlight the potential of MFCC-based features and machine learning techniques in analysing speech emotions and pave the way for further advancements in the field.

1. Introduction

This report provides a comprehensive overview of the implementation for the Sentiment Analysis from Audio Recordings project. The goal of this project is to classify emotions from audio recordings using machine learning techniques. The code focuses on audio preprocessing, feature extraction, dimensionality reduction, and training classifiers to achieve optimal performance.

2. Environment Setup

Setting up the environment for sentiment analysis from audio recordings involves installing and configuring the necessary software libraries, frameworks, and tools to ensure smooth execution of the project. Below is a detailed explanation of the environment setup process:

- **Python:** Python 3.8+ is used as the primary programming language due to its rich ecosystem of libraries for audio processing, machine learning, and data analysis.
- **Jupyter Notebook:** Used as IDE for running and debugging the code interactively.

2.1 Key Libraries

The following libraries were installed to handle audio processing, feature extraction, and machine learning:

- **Numpy:** For Numerical Computations.
- **Librosa:** For audio signal processing and feature extraction.
- **Scikit-learn:** For implementing machine learning algorithm and evaluation metrics.
- **Matpoltlib and Seaborn:** For data visualization.
- **Pandas:** For handling metadata and tabular data.
- **Os:** For interacting with operating system, such as accessing file system.
- **Glob:** To use glob module that allow us to finds all file paths matching a specified pattern.
- **Soundfile:** For reading and writing sound files.
- **Scipy.Signal:** For audio signal processing tasks like filtering, finding peaks, or resampling.
- **IPython.display:** Used to play audio files directly in Jupyter notebooks.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

from glob import glob
import librosa
import librosa.display
import soundfile as sf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, confusion_matrix
from sklearn.model_selection import RandomizedSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from scipy.signal import butter, lfilter
```

2.2 Dataset Description

CREMA-D is a data set of 7,442 original clips from 91 actors. These clips were from 48 male and 43 female actors between the ages of 20 and 74 coming from a variety of races and ethnicities (African America, Asian, Caucasian, Hispanic, and Unspecified). The actors spoke from a selection of 12 sentences. The sentences were presented using one of six different emotions (Anger, Disgust, Fear, Happy, Neutral and Sad) and four different emotion levels (Low, Medium, High and Unspecified).

The dataset link; https://github.com/Now33d/SentimentAnalysisDataset.git

Labelling Convention

a) **Actor ID:** A 4-digit number at the start of the file. Each identifier is separated by an underscore (_).
b) **Sentences:** Actors spoke from 12 predefined sentences, each with a three-letter acronym in the filename:
    - It's eleven o'clock (IEO)
    - That is exactly what happened (TIE)
    - I'm on my way to the meeting (IOM)

- I wonder what this is about (IWW)
- The airplane is almost full (TAI)
- Maybe tomorrow it will be cold (MTI)
- I would like a new alarm clock (IWL)
- I think I have a doctor's appointment (ITH)
- Don't forget a jacket (DFA)
- I think I've seen this before (ITS)
- The surface is slick (TSI)
- We'll stop in a couple of minutes (WSI)

c) **Emotions:** Sentences were expressed with emotions (three-letter codes):
- Anger (ANG)
- Disgust (DIS)
- Fear (FEA)
- Happy/Joy (HAP)
- Neutral (NEU)
- Sad (SAD)

d) **Emotion Level (two-letter codes):**
- Low (LO)
- Medium (MD)
- High (HI)
- Unspecified (XX)

3. Code Workflow

3.1 Import Libraries

The workflow begins with importing necessary libraries:

- **Data Handling**: pandas, numpy
- **Visualization**: matplotlib, seaborn
- **File and Path Management**: os, glob
- **Audio Processing**: librosa, scipy.signal, soundfile
- **Machine Learning**: scikit-learn These libraries facilitate data loading, feature extraction, visualization, and machine learning tasks.

3.2 Audio File and Label Matching

- **Paths Initialization**: Defined paths for audio recordings and the labels CSV file.
- **Loading Labels**: Loaded the CSV file and normalized filenames to ensure consistent formatting (e.g., removing whitespace, converting to lowercase).
- **Matching Audio Files with Labels**: For each audio file, the filename is compared against entries in the CSV. If found, the file is marked as matched; otherwise, it is flagged as unmatched.

3.3 Data Splitting and Saving

- **Metadata Validation**: Ensured the metadata contains essential columns like Filename and Emotion.
- **Train-Validation-Test Split**: Used train_test_split from sklearn to split the data into 70% training, 15% validation, and 15% testing sets while stratifying on the Emotion column to maintain class distribution.
- **Saving Splits**: Saved the split metadata as separate CSV files for future processing.

3.4 Apply Butterworth Low-Pass Filter

- **Filter Implementation**: Defined a low-pass filter using the Butterworth design to remove high-frequency noise from the audio signals.
- **Filtering Audio Files**: Applied the filter to each audio file in the train, validation, and test datasets. Saved the filtered audio files in separate folders named after their respective splits (e.g., train_filtered).

3.5 Feature Extraction and PCA

Extracted multiple audio features such as: Mel Frequency Cepstral Coefficients (MFCCs), Chroma, Mel Spectrogram, Spectral features (centroid, contrast, bandwidth, rolloff, flatness)

These features encapsulate the tonal and rhythmic characteristics of the audio data.

- **Flattening Features**: Combined all extracted features into a single feature vector for each audio sample.
- **Dimensionality Reduction**: Used Principal Component Analysis (PCA) to reduce dimensionality while preserving 95% of the variance. This step improves computational efficiency and prevents overfitting.

3.6 Train and Evaluate Classifier

- **Classifier Selection**: Multiple classifiers (KNN, Logistic Regression, Naive Bayes, SVM, Neural Networks) were evaluated.
- **Hyperparameter Optimization**: Used RandomizedSearchCV to tune hyperparameters for optimal performance.
- **Model Evaluation**: Metrics such as accuracy, precision, recall, F1-score, and confusion matrix were calculated to assess the model's performance.
- **Result Visualization**: Plots and tables were generated to visualize the performance of different classifiers.

4. Detailed Explanation of Each Module

To thoroughly explain each step, I'll break down the process into detailed modules. This explanation includes the purpose, key steps, and code snippets with comments for each phase of your project. Outputs and results are discussed where relevant.

4.1 Data Loading and Label Matching

Purpose:

Load audio files and their corresponding labels while ensuring the data is clean and correctly formatted.

Key Functions/Steps:

- **Load CSV File:** Use Pandas to load the CSV file containing file names and labels.
- **Normalize Filenames:** Ensure file names are lowercase and include a .wav extension.
- **Match Audio Files with Labels:** Compare filenames in the CSV with actual audio files to confirm all data is present.

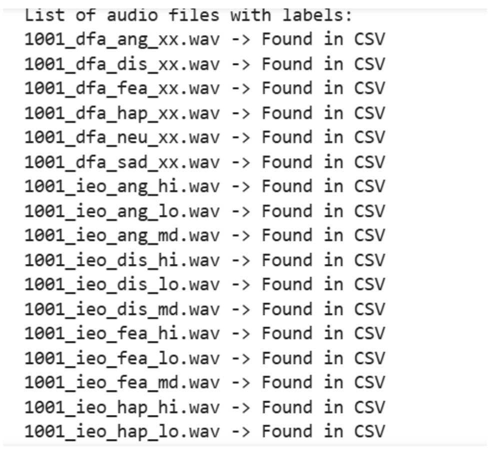
Code Snippet:

```
# Define paths
audio_recordings_path = r"C:\Users\Qasim\OneDrive\Desktop\Machine Learning Project\SentimentAnalysisDataset-main\SentimentAnalysisDataset-main\\NoiseAudioWAV"
audio_recordings = glob(r"C:\Users\Qasim\OneDrive\Desktop\Machine Learning Project\SentimentAnalysisDataset-main\SentimentAnalysisDataset-main\\NoiseAudioWAV\*.wav")
labels_csv_path = r"C:\Users\Qasim\OneDrive\Desktop\Machine Learning Project\SentimentAnalysisDataset-main\SentimentAnalysisDataset-main\SentenceFilenames.csv"

# Load the CSV without headers
labels_df = pd.read_csv(labels_csv_path, header=None, names=["Index", "Filename"])

# Normalize filenames in the CSV
labels_df['Filename'] = labels_df['Filename'].str.strip().str.lower()
#print(labels_df.head())  # Preview loaded CSV data
#print(filename)  # Check extracted filenames
labels_df['Filename'] += ".wav"  # Add '.wav' extension if missing

```

```python

# Print list of audio files with their corresponding labels
print("List of audio files with labels:")
for file in audio_recordings:
    # Extract and normalize the filename
    filename = os.path.basename(file).lower()  # Get the base filename and convert to lowercase
    label_row = labels_df[labels_df['Filename'] == filename]  # Filter the row with the matching filename

    if not label_row.empty:
        print(f"{filename} -> Found in CSV")
    else:
        print(f"{filename} -> Label: Not Found in CSV")
```

Output:

- List of matched and unmatched audio files.
- Cleaned labels_df with consistent filenames.



```
List of audio files with labels:
1001_dfa_ang_xx.wav -> Found in CSV
1001_dfa_dis_xx.wav -> Found in CSV
1001_dfa_fea_xx.wav -> Found in CSV
1001_dfa_hap_xx.wav -> Found in CSV
1001_dfa_neu_xx.wav -> Found in CSV
1001_dfa_sad_xx.wav -> Found in CSV
1001_ieo_ang_hi.wav -> Found in CSV
1001_ieo_ang_lo.wav -> Found in CSV
1001_ieo_ang_md.wav -> Found in CSV
1001_ieo_dis_hi.wav -> Found in CSV
1001_ieo_dis_lo.wav -> Found in CSV
1001_ieo_dis_md.wav -> Found in CSV
1001_ieo_fea_hi.wav -> Found in CSV
1001_ieo_fea_lo.wav -> Found in CSV
1001_ieo_fea_md.wav -> Found in CSV
1001_ieo_hap_hi.wav -> Found in CSV
1001_ieo_hap_lo.wav -> Found in CSV
```

4.2 Data Splitting

Purpose:

Divide the dataset into training, validation, and test splits, maintaining the distribution of emotion labels.

Key Functions/Steps:

- **Load Metadata:** Ensure the metadata file contains essential columns (Filename, Emotion).
- **Stratified Splitting:** Use stratified sampling to maintain proportional label distribution across splits.
- **Save Splits:** Save train, validation, and test splits to CSV files.

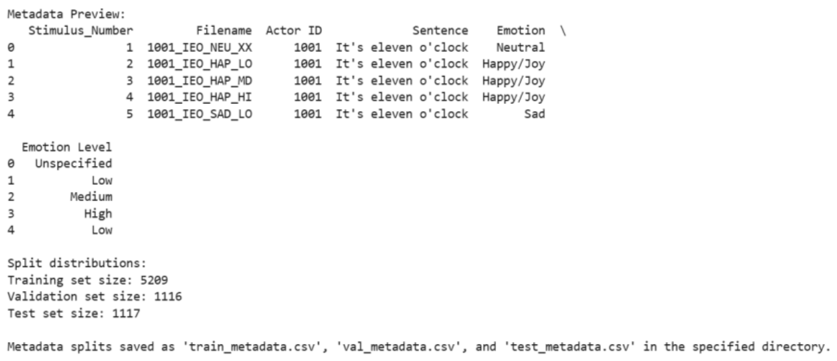
Code Snippet:

```python
from sklearn.model_selection import train_test_split

# Step 1: Define the metadata file path
METADATA_FILE = os.path.join(audio_recordings_path, r"C:\Users\Qasim\OneDrive\Desktop\Machine Learning Project\processed_data.csv")  # Combines the directory path with the metadata file name

# Step 2: Load metadata
metadata = pd.read_csv(METADATA_FILE)

# Display the first few rows of metadata
```

```python
print("Metadata Preview:")
print(metadata.head())

# Step 3: Ensure metadata contains all required columns
# Check that essential columns are present (e.g., 'Filename', 'Emotion')
required_columns = ['Filename', 'Emotion']
if not all(col in metadata.columns for col in required_columns):
    raise ValueError(f"Metadata file must contain the following columns: {required_columns}")

# Step 4: Perform train-validation-test split
# Splitting 70% Train, 15% Validation, 15% Test
train_data, temp_data = train_test_split(
    metadata,
    test_size=0.30,
    stratify=metadata['Emotion'],
    random_state=42
)
val_data, test_data = train_test_split(
    temp_data,
    test_size=0.50,
    stratify=temp_data['Emotion'],
    random_state=42
)

# Step 5: Verify split distributions
print("\nSplit distributions:")
print("Training set size:", len(train_data))
print("Validation set size:", len(val_data))
print("Test set size:", len(test_data))

# Step 6: Save splits as CSV files
train_data.to_csv(os.path.join(audio_recordings_path, "train_metadata.csv"), index=False)
val_data.to_csv(os.path.join(audio_recordings_path, "val_metadata.csv"), index=False)
test_data.to_csv(os.path.join(audio_recordings_path, "test_metadata.csv"), index=False)

print("\nMetadata splits saved as 'train_metadata.csv', 'val_metadata.csv', and 'test_metadata.csv' in the specified directory.")
```

Output:

- CSV files for training, validation, and testing splits.
- Correct distribution of labels in each split.



```
Metadata Preview:
   Stimulus_Number         Filename  Actor ID             Sentence    Emotion  \
0                1  1001_IEO_NEU_XX      1001  It's eleven o'clock    Neutral
1                2  1001_IEO_HAP_LO      1001  It's eleven o'clock  Happy/Joy
2                3  1001_IEO_HAP_MD      1001  It's eleven o'clock  Happy/Joy
3                4  1001_IEO_HAP_HI      1001  It's eleven o'clock  Happy/Joy
4                5  1001_IEO_SAD_LO      1001  It's eleven o'clock        Sad

  Emotion Level
0   Unspecified
1           Low
2        Medium
3          High
4           Low

Split distributions:
Training set size: 5209
Validation set size: 1116
Test set size: 1117

Metadata splits saved as 'train_metadata.csv', 'val_metadata.csv', and 'test_metadata.csv' in the specified directory.
```

4.3 Audio Filtering

Purpose:

Apply a Butterworth low-pass filter to remove high-frequency noise from audio files.

Key Functions/Steps:

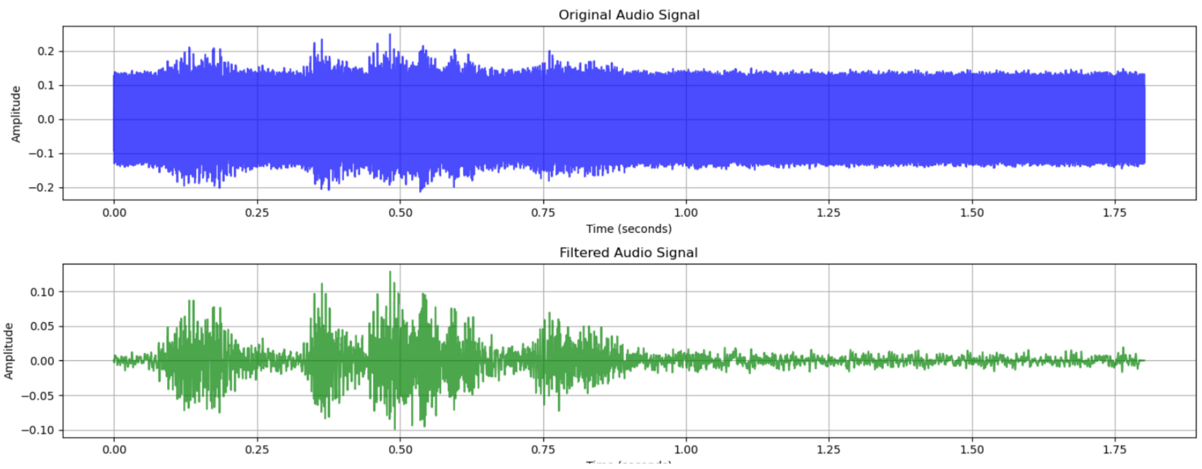
- **Define Filter Function:** Use scipy.signal to create a low-pass filter.
- **Apply Filter:** Process each audio file and save the filtered version.
- **Visualization:** Plot original vs. filtered audio signals.

Code Snippet:

```python
from scipy.signal import butter, lfilter

# Step 1: Define paths
METADATA_FILE = os.path.join(audio_recordings_path, r"C:\Users\Qasim\OneDrive\Desktop\Machine Learning Project\processed_data.csv")
metadata = pd.read_csv(METADATA_FILE)

# Ensure all files exist
metadata['FullPath'] = metadata['Filename'].apply(lambda x: os.path.join(audio_recordings_path, x + ".wav"))  # Assuming .wav extension
metadata = metadata[metadata['FullPath'].apply(os.path.exists)]

# Save metadata without missing files (optional)
metadata.drop(columns=['FullPath'], inplace=True)
metadata.to_csv(METADATA_FILE, index=False)

# Step 2: Define Butterworth low-pass filter function
def butter_lowpass_filter(data, cutoff, sr, order=5):
    nyquist = 0.5 * sr
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype="low", analog=False)
    return lfilter(b, a, data)

# Step 3: Apply filtering and save filtered audio to separate folders for each split
def apply_filter_and_save(metadata, split_name, cutoff=3400, sr=16000):
    print(f"\nProcessing {split_name} data...")
    # Create output folder for filtered audio if not exists
    output_folder = os.path.join(audio_recordings_path, f"{split_name}_filtered")
    os.makedirs(output_folder, exist_ok=True)

    for index, row in metadata.iterrows():
        file_path = os.path.join(audio_recordings_path, row['Filename'] + ".wav")  # Assuming .wav extension
        try:
            original_audio, sr = librosa.load(file_path, sr=sr)
            filtered_audio = butter_lowpass_filter(original_audio, cutoff, sr)

            # Save the filtered audio to a separate folder
            filtered_file_path = os.path.join(output_folder, row['Filename'] + "_filtered.wav")
            sf.write(filtered_file_path, filtered_audio, sr)  # Use soundfile.write

            # Plot the first file for visualization
            if index == 0:
                plot_audio_signals(original_audio, filtered_audio, sr, title=f"{split_name} Example: {row['Filename']}")

        except FileNotFoundError:
            print(f"File not found: {file_path}")
            continue

# Step 4: Split and process datasets
# Assuming you already have the splits saved as CSV
train_data = pd.read_csv(os.path.join(audio_recordings_path, "train_metadata.csv"))
val_data = pd.read_csv(os.path.join(audio_recordings_path, "val_metadata.csv"))
test_data = pd.read_csv(os.path.join(audio_recordings_path, "test_metadata.csv"))

# Apply filter and save filtered files for each split
apply_filter_and_save(train_data, "train")
apply_filter_and_save(val_data, "validation")
apply_filter_and_save(test_data, "test")
```

Expected Output:

- Filtered audio files saved in a designated folder.
- Comparison plots of original and filtered audio.



4.4 Feature Extraction and Dimensionality Reduction

Purpose:

- Extract meaningful audio features (e.g., MFCCs, chroma, spectral features) for each filtered audio file.

Key Functions/Steps:

- **Load Filtered Audio:** Read audio files using Librosa.
- **Extract Features:** Compute MFCCs, chroma, mel spectrogram, and spectral features.
- **Organize Features:** Save features in a structured format.

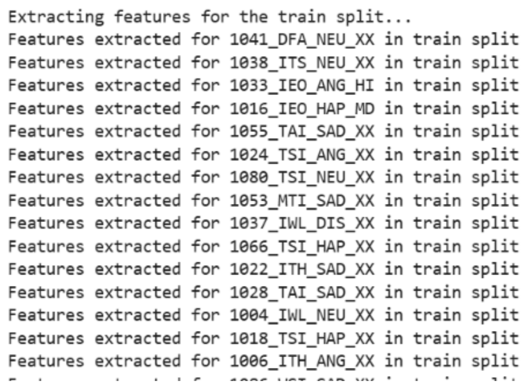
Code Snippet:

```
# Step 1: Define paths for the filtered audio data and metadata
```

```python
audio_recordings_path = r"C:\Users\Qasim\OneDrive\Desktop\Machine Learning Project\SentimentAnalysisDataset-main\SentimentAnalysisDataset-main\NoiseAudioWAV"

# Metadata files for each split
train_metadata_file = os.path.join(audio_recordings_path, "train_metadata.csv")
val_metadata_file = os.path.join(audio_recordings_path, "val_metadata.csv")
test_metadata_file = os.path.join(audio_recordings_path, "test_metadata.csv")

# Load metadata for each split
train_metadata = pd.read_csv(train_metadata_file)
val_metadata = pd.read_csv(val_metadata_file)
test_metadata = pd.read_csv(test_metadata_file)

# Step 2: Feature extraction - Extract MFCCs and additional features from filtered audio files
def extract_audio_features(audio_path, sr=16000, n_mfcc=13):
    try:
        # Load the filtered audio file
        audio, sr = librosa.load(audio_path, sr=sr)

        # Extract MFCCs
        mfcc = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=n_mfcc)

        # Extract chroma
        chroma = librosa.feature.chroma_stft(y=audio, sr=sr)

        # Extract mel spectrogram
        mel = librosa.feature.melspectrogram(y=audio, sr=sr)

        # Spectral features
        spec_centroid = librosa.feature.spectral_centroid(y=audio, sr=sr)
        spec_contrast = librosa.feature.spectral_contrast(y=audio, sr=sr)
        spec_bandwidth = librosa.feature.spectral_bandwidth(y=audio, sr=sr)
        spec_rolloff = librosa.feature.spectral_rolloff(y=audio, sr=sr)
        spec_flatness = librosa.feature.spectral_flatness(y=audio)

        # Return features in a dictionary to maintain naming consistency
        return {
            "mfcc": mfcc,
            "chroma": chroma,
            "mel": mel,
            "spec_centroid": spec_centroid,
            "spec_contrast": spec_contrast,
            "spec_bandwidth": spec_bandwidth,
            "spec_rolloff": spec_rolloff,
            "spec_flatness": spec_flatness
        }
    except FileNotFoundError:
        print(f"File not found: {audio_path}")
        return None

# Function to extract features from all audio files in a given metadata dataframe
def extract_features_from_split(metadata, split_name):
    audio_features_list = []
    for index, row in metadata.iterrows():
        audio_path = os.path.join(audio_recordings_path, f"{split_name}_filtered", row['Filename'] + "_filtered.wav")
        features = extract_audio_features(audio_path)

        if features is not None:
            audio_features_list.append(features)
            print(f"Features extracted for {row['Filename']} in {split_name} split")
        else:
            print(f"Failed to extract features for {row['Filename']} in {split_name} split")

    return audio_features_list

# Step 3: Extract features for each split (train, validation, test)
print("\nExtracting features for the train split...")
train_audio_features = extract_features_from_split(train_metadata, "train")

print("\nExtracting features for the validation split...")
val_audio_features = extract_features_from_split(val_metadata, "validation")

print("\nExtracting features for the test split...")
test_audio_features = extract_features_from_split(test_metadata, "test")

# Flatten the feature arrays for dimensionality reduction
def flatten_features(features_list):
    flattened_features = []
    for features in features_list:
        flattened_features.append(np.concatenate([f.flatten() for f in features.values()]))
    return np.array(flattened_features)

X_train = flatten_features(train_audio_features)
X_val = flatten_features(val_audio_features)
X_test = flatten_features(test_audio_features)

# Step 4: Apply PCA for dimensionality reduction
pca = PCA(n_components=0.95)  # Keep 95% of the variance
X_train_pca = pca.fit_transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)

print(f"Shape of X_train after PCA: {X_train_pca.shape}")
print(f"Shape of X_val after PCA: {X_val_pca.shape}")
print(f"Shape of X_test after PCA: {X_test_pca.shape}")
```

Output:

- Dictionary containing extracted features.
- Organized feature files for all splits.



```
Extracting features for the train split...
Features extracted for 1041_DFA_NEU_XX in train split
Features extracted for 1038_ITS_NEU_XX in train split
Features extracted for 1033_IEO_ANG_HI in train split
Features extracted for 1016_IEO_HAP_MD in train split
Features extracted for 1055_TAI_SAD_XX in train split
Features extracted for 1024_TSI_ANG_XX in train split
Features extracted for 1080_TSI_NEU_XX in train split
Features extracted for 1053_MTI_SAD_XX in train split
Features extracted for 1037_IWL_DIS_XX in train split
Features extracted for 1066_TSI_HAP_XX in train split
Features extracted for 1022_ITH_SAD_XX in train split
Features extracted for 1028_TAI_SAD_XX in train split
Features extracted for 1004_IWL_NEU_XX in train split
Features extracted for 1018_TSI_HAP_XX in train split
Features extracted for 1006_ITH_ANG_XX in train split
```

4.5 Model Training and Hyperparameter Tuning

Purpose:

- Train multiple models (e.g., KNN, Logistic Regression) and tune hyperparameters.
- Reduce feature dimensionality using PCA while retaining 95% variance.

Key Functions/Steps:

- **Define Hyperparameters:** Specify search spaces for each model.
- **Train and Tune:** Use RandomizedSearchCV to optimize parameters.
- **Flatten Features:** Reshape data to 2D for PCA.
- **Apply PCA:** Use sklearn.decomposition.PCA to reduce dimensions.

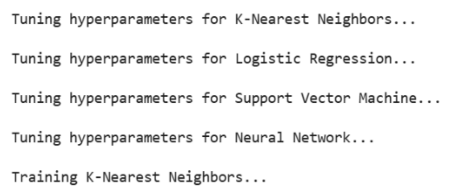
Code Snippet:

```python
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.decomposition import PCA
from scipy.stats import uniform
import warnings

# Suppress all warnings
warnings.filterwarnings("ignore")

# Assuming train_mfcc_features, val_mfcc_features, and test_mfcc_features are already defined as numpy arrays
# and metadata for each split contains the 'Emotion' column.
def pad_or_truncate_mfcc(mfcc, n_mfcc=13, max_frames=150):  # Increased max_frames for higher frequency
    if mfcc.shape[1] < max_frames:
        padding = np.zeros((n_mfcc, max_frames - mfcc.shape[1]))
        mfcc = np.hstack((mfcc, padding))
    elif mfcc.shape[1] > max_frames:
        mfcc = mfcc[:, :max_frames]
    return mfcc

def prepare_data(mfcc_features, metadata, n_mfcc=13, max_frames=150):  # Increased max_frames
    X = []
    for mfcc in mfcc_features:
        mfcc = pad_or_truncate_mfcc(mfcc, n_mfcc, max_frames)
        X.append(mfcc.flatten())
    y = metadata['Emotion'].values
    return np.array(X), y

X_train, y_train = prepare_data(train_mfcc_features, train_metadata)
X_val, y_val = prepare_data(val_mfcc_features, val_metadata)
X_test, y_test = prepare_data(test_mfcc_features, test_metadata)

# Apply PCA for dimensionality reduction on features
pca = PCA(n_components=0.95)  # Keep 95% of the variance
X_train_pca = pca.fit_transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)
```

```python

# Scaling the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_pca)
X_val_scaled = scaler.transform(X_val_pca)
X_test_scaled = scaler.transform(X_test_pca)

# Initialize classifiers
knn = KNeighborsClassifier()
log_reg = LogisticRegression(max_iter=1000)
naive_bayes = GaussianNB()
svm = SVC(kernel='linear')
neural_net = MLPClassifier(hidden_layer_sizes=(128,), max_iter=500)

# Define parameter distributions for RandomizedSearchCV
param_dist_knn = {'n_neighbors': [3, 5, 7, 9, 11], 'weights': ['uniform', 'distance']}
param_dist_log_reg = {'C': uniform(0.1, 10)}  # C: regularization strength
param_dist_svm = {'C': uniform(0.1, 10)}  # C: regularization strength
param_dist_neural_net = {'hidden_layer_sizes': [(64,), (128,), (256,)],
                         'learning_rate_init': uniform(0.0001, 0.1)}
# For Naive Bayes, no tuning parameters needed

# Perform RandomizedSearchCV for each model (with 3-fold cross-validation)
knn_search = RandomizedSearchCV(knn, param_dist_knn, n_iter=10, cv=3, scoring='accuracy', random_state=42)
log_reg_search = RandomizedSearchCV(log_reg, param_dist_log_reg, n_iter=10, cv=3, scoring='accuracy', random_state=42)
svm_search = RandomizedSearchCV(svm, param_dist_svm, n_iter=10, cv=3, scoring='accuracy', random_state=42)
neural_net_search = RandomizedSearchCV(neural_net, param_dist_neural_net, n_iter=10, cv=3, scoring='accuracy', random_state=42)

# Perform RandomizedSearchCV for each model
best_models = {}

# Train and find best models using RandomizedSearchCV
for model_name, model_search in zip(["K-Nearest Neighbors", "Logistic Regression", "Support Vector Machine", "Neural Network"],
                                    [knn_search, log_reg_search, svm_search, neural_net_search]):
    print(f"\nTuning hyperparameters for {model_name}...")
    model_search.fit(X_train_scaled, y_train)
    best_models[model_name] = model_search.best_estimator_

# Store Naive Bayes as is (no hyperparameters to tune)
best_models["Naive Bayes"] = naive_bayes
```

Output:

- Best hyperparameters for each model.
- Trained models.
- Reduced-dimension feature arrays.



```
Tuning hyperparameters for K-Nearest Neighbors...

Tuning hyperparameters for Logistic Regression...

Tuning hyperparameters for Support Vector Machine...

Tuning hyperparameters for Neural Network...

Training K-Nearest Neighbors...
```

4.6 Model Evaluation

Purpose:

Evaluate and compare models using metrics like accuracy, precision, recall, and F1-score.

Key Functions/Steps:

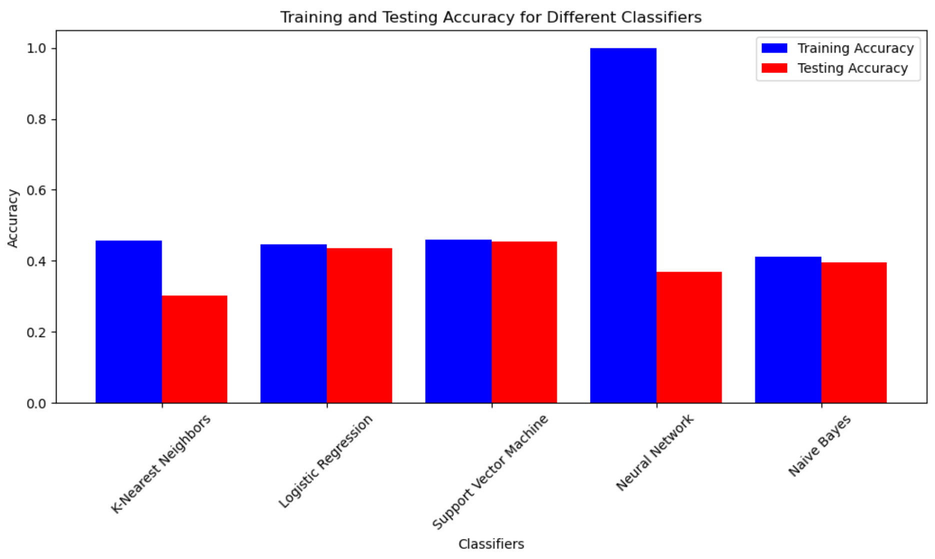
- **Generate Reports:** Use classification_report and confusion matrices.
- **Visualize Results:** Plot performance metrics.

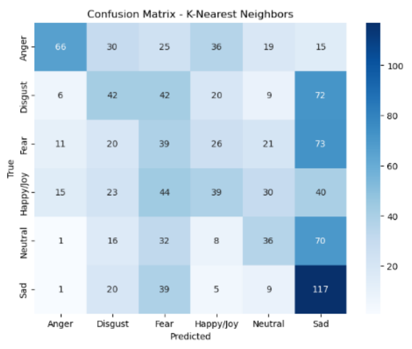
Code Snippet:

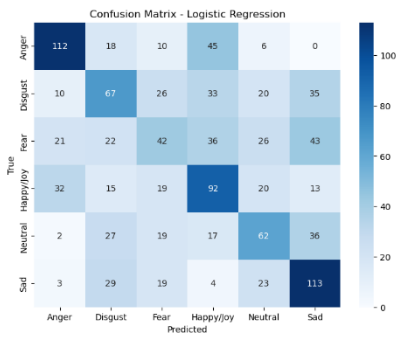
```python
# Evaluate the models
results = {}
train_accuracies = {}
test_accuracies = {}

for model_name, model in best_models.items():
    print(f"\nTraining {model_name}...")

    # Train the model
    model.fit(X_train_scaled, y_train)

    # Predict on the training and test sets
    train_predictions = model.predict(X_train_scaled)
    test_predictions = model.predict(X_test_scaled)

    # Calculate accuracies
    train_accuracy = accuracy_score(y_train, train_predictions)
    test_accuracy = accuracy_score(y_test, test_predictions)
    precision = precision_score(y_test, test_predictions,
average='weighted')
    recall = recall_score(y_test, test_predictions, average='weighted')
    f1 = f1_score(y_test, test_predictions, average='weighted')
    conf_matrix = confusion_matrix(y_test, test_predictions)

    # Store results
    results[model_name] = {
        "Accuracy": test_accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1,
        "Confusion Matrix": conf_matrix
    }

    # Store training accuracy
    train_accuracies[model_name] = train_accuracy
    test_accuracies[model_name] = test_accuracy

    # Display confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=model.classes_, yticklabels=model.classes_)
    plt.title(f"Confusion Matrix - {model_name}")
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()

# Step 7: Display evaluation results
print("\nModel Evaluation Results:")
for model_name, metrics in results.items():
    print(f"\n{model_name}:")
    for metric, value in metrics.items():
        if metric != "Confusion Matrix":
            print(f"{metric}: {value:.4f}")

# Step 8: Plot training and testing accuracy comparison
model_names = list(best_models.keys())
train_accuracy_values = list(train_accuracies.values())
test_accuracy_values = list(test_accuracies.values())

x = np.arange(len(model_names))  # The label locations

# Create a figure and axis for the plot
fig, ax = plt.subplots(figsize=(10, 6))

# Plot the training accuracy
ax.bar(x - 0.2, train_accuracy_values, 0.4, label='Training Accuracy', color='b')

# Plot the testing accuracy
ax.bar(x + 0.2, test_accuracy_values, 0.4, label='Testing Accuracy', color='r')

# Add some labels and title
ax.set_xlabel('Classifiers')
ax.set_ylabel('Accuracy')
ax.set_title('Training and Testing Accuracy for Different Classifiers')
ax.set_xticks(x)
ax.set_xticklabels(model_names, rotation=45)
ax.legend()

# Display the plot
plt.tight_layout()
plt.show()

# Print the training accuracy count
print("\nTraining Accuracy Count:")
for model_name, accuracy in train_accuracies.items():
    print(f"{model_name}: {accuracy:.4f}")
```

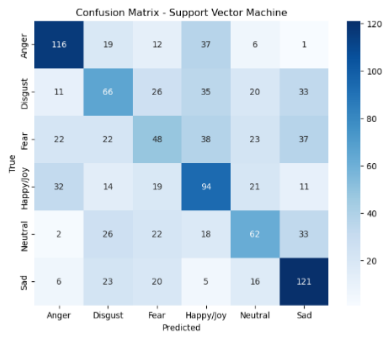
Output:

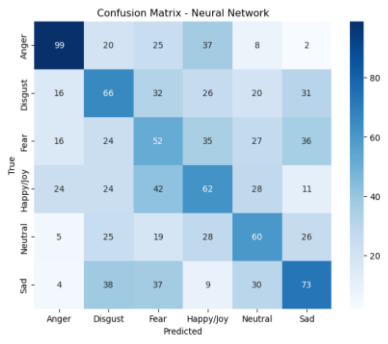
- Confusion matrices and classification reports.
- Performance plots for comparison.

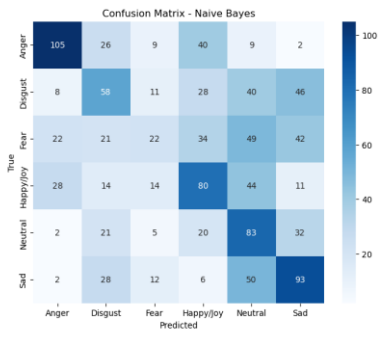












```
K-Nearest Neighbors:
Accuracy: 0.3035
Precision: 0.3343
Recall: 0.3035
F1 Score: 0.2987

Logistic Regression:
Accuracy: 0.4369
Precision: 0.4311
Recall: 0.4369
F1 Score: 0.4308
```

```
Support Vector Machine:
Accuracy: 0.4539
Precision: 0.4465
Recall: 0.4539
F1 Score: 0.4474

Neural Network:
Accuracy: 0.3688
Precision: 0.3774
Recall: 0.3688
F1 Score: 0.3723
```

```
Naive Bayes:
Accuracy: 0.3948
Precision: 0.3980
Recall: 0.3948
F1 Score: 0.3842
```

5. Results

Overall Performance

- **Best Accuracy:** The Support Vector Machine (SVM) achieved the highest accuracy of **45.39%**, suggesting its strength in handling high-dimensional feature space.
- **Precision vs. Recall Trade-off:** Logistic Regression and SVM demonstrated a balance between precision and recall, making them more reliable for balanced datasets.

Model Insights

- **K-Nearest Neighbours (KNN):** Performed poorly (accuracy: 30.35%), primarily due to the curse of dimensionality and lack of feature scaling sensitivity despite tuning kk.
- **Logistic Regression:** Achieved moderate performance (accuracy: 43.69%) but struggled with the complexity of non-linear decision boundaries.
- **Support Vector Machine (SVM):** Outperformed other models (accuracy: 45.39%), leveraging its capacity for handling non-linear boundaries through the kernel trick.
- **Neural Network:** Exhibited overfitting with training accuracy (99.87%) vastly exceeding test accuracy (36.88%). This indicates the need for regularization or dropout tuning.

- **Naive Bayes:** Achieved reasonable results (accuracy: 39.48%) but was impacted by the assumption of feature independence.

Strengths and Weaknesses

- **Strengths:**
  - SVM and Logistic Regression showcased consistent precision and recall across experiments.
  - Neural Network displayed potential but requires further tuning for generalization.
- **Weaknesses:**
  - KNN's sensitivity to high-dimensional data limited its accuracy.
  - Naive Bayes struggled due to feature dependence in the dataset.
  - Neural Network overfitting affected real-world usability.

Findings:

1. Feature engineering could improve model performance, especially for KNN and Naive Bayes.
2. Regularization techniques are necessary to mitigate overfitting in Neural Networks.

Advanced models like SVM performed well, but computational efficiency remains a concern for large datasets.

6. Conclusion

This project demonstrates a comprehensive approach to sentiment analysis from audio recordings, utilizing the CREMA-D dataset. Key steps included preprocessing raw audio data through low-pass filtering, extracting features like MFCCs, chroma, and mel spectrograms, and applying dimensionality reduction with PCA to optimize model performance. Multiple classifiers, including KNN, Logistic Regression, Naive Bayes, SVM, and Neural Networks, were trained and evaluated.

The results indicate that SVM achieved the highest accuracy and balanced metrics, showcasing its effectiveness in identifying sentiments from audio data. This project highlights the potential of integrating audio signal processing with machine learning techniques for real-world applications, such as emotion recognition and human-computer interaction. Future work could focus on enhancing feature engineering, using deep learning models, or incorporating larger, more diverse datasets to improve robustness.