Audio Clustering Track 1

Rohit Yuvaraj Jadekar
Entry No.: 2022MT61984

Akarsh Gupta
Entry No.: 2022MT61966

April 12, 2025

1 Non-Comp Part

This report analyzes the shared code implementation for audio signal processing and the corresponding waveform visualizations.

1.1 Code Analysis and Framework

The provided code implements a complete audio processing pipeline for clustering and classification. The implementation follows a structured approach with several key components:

1.1.1 Setup and Library Configuration

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

# Fix numpy-librosa compatibility
import numpy as np
np.complex = complex
```

The code begins by mounting Google Drive for data access and resolving a compatibility issue between NumPy and Librosa libraries. This is a common workaround when working with audio processing libraries in Colab environments.

2 Feature Extraction from Audio Files

The code extracts meaningful features from audio files, condensing raw audio signals into descriptive numerical representations for further analysis. Below is a summary of the process:

2.1 Feature Extraction Function

```
# STEP 1: FEATURE EXTRACTION
import librosa
import numpy as np
from scipy.stats import skew, kurtosis

import librosa
import numpy as np

def extract_features(file_path):
    y, sr = librosa.load(file_path, sr=None)
    features = {}

    # --- MFCCs ---
    mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=20)
    mfccs_delta = librosa.feature.delta(mfccs)
    mfccs_delta2 = librosa.feature.delta(mfccs, order=2)

    for i in range(20):
        features[f'mfcc_{i+1}_mean'] = np.mean(mfccs[i])
        features[f'mfcc_{i+1}_var'] = np.var(mfccs[i])
        features[f'delta_{i+1}_mean'] = np.mean(mfccs_delta[i])
        features[f'delta2_{i+1}_mean'] = np.mean(mfccs_delta2[i])

    # --- Zero Crossing Rate ---
    zcr = librosa.feature.zero_crossing_rate(y)[0]
    features['zcr_mean'] = np.mean(zcr)
    features['zcr_var'] = np.var(zcr)

    # --- Spectral Centroid ---
    sc = librosa.feature.spectral_centroid(y=y, sr=sr)[0]
    features['spec_centroid_mean'] = np.mean(sc)
    features['spec_centroid_var'] = np.var(sc)

    # --- Spectral Bandwidth ---
    bw = librosa.feature.spectral_bandwidth(y=y, sr=sr)[0]
    features['spec_bw_mean'] = np.mean(bw)
    features['spec_bw_var'] = np.var(bw)

    # --- Root Mean Square Energy ---
    rms = librosa.feature.rms(y=y)[0]
    features['rms_mean'] = np.mean(rms)
    features['rms_var'] = np.var(rms)

    # --- Tempo and IOI (Inter-Onset Interval) ---
    onset_env = librosa.onset.onset_strength(y=y, sr=sr)
    onset_frames = librosa.onset.onset_detect(onset_envelope=onset_env, sr=
    sr)
    onset_times = librosa.frames_to_time(onset_frames, sr=sr)
```

```
    iois = np.diff(onset_times)

    features['tempo'] = librosa.beat.tempo(onset_envelope=onset_env, sr=sr)[0]
    features['ioi_mean'] = np.mean(iois) if len(iois) > 0 else 0
    features['ioi_var'] = np.var(iois) if len(iois) > 0 else 0

    return features
```

The `extract_features(file_path)` function performs the following operations:

1. **MFCCs and Derivatives:** Computes 20 MFCCs (Mel-Frequency Cepstral Coefficients) along with their first-order (delta) and second-order (delta-delta) derivatives. These capture timbral properties and spectral changes over time.

2. **Why Use 20 MFCCs?**

   MFCCs (Mel-Frequency Cepstral Coefficients) capture the short-term spectral shape of an audio signal using the Mel-scaled power spectrum and Discrete Cosine Transform (DCT).

2.2 Rationale for 20 MFCCs

- **Lower MFCCs (1st–4th):** Capture energy, loudness, and pitch.
- **Middle MFCCs (5th–13th):** Represent timbre, voice type, and tone.
- **Higher MFCCs (14th onward):** Add finer spectral details but are prone to noise.

Using 20 MFCCs balances broad tone representation with moderate fine-grain detail, making it ideal for non-speech audio with diverse timbral variations like snoring or rain.

3. **Zero Crossing Rate (ZCR):** Measures how often the signal changes sign, useful for detecting noisiness and tonal quality.

4. **Spectral Features:** Computes the mean and variance of spectral centroid (brightness) and spectral bandwidth (frequency spread).

5. **Root Mean Square (RMS) Energy:** Quantifies signal power (loudness) over time.

6. **Tempo and Inter-Onset Interval (IOI):** Detects onset times, computes inter-onset intervals, and estimates tempo to capture rhythmic properties.

2.3 Batch Processing

The function is applied to all audio files listed in `df_labels`. For each file:

- Features are extracted and stored in a dictionary.
- Metadata such as filename and category are appended.
- Results are saved to a CSV file for further analysis.

2.4 Conceptual Overview

This process condenses raw audio data into perceptually relevant features, enabling machine learning tasks like classification or clustering. Key benefits include:

1. Dimensionality reduction from raw signals to meaningful statistics.
2. Statistical characterization of audio attributes (mean, variance).
3. Capturing rhythmic properties through tempo and IOI analysis.

2.5 Waveform Analysis

The code includes visualization of waveforms with:

```
sample_paths = df_labels.iloc[:900].reset_index(drop=True)['filename'].iloc[:5].apply(
    lambda fn: os.path.join(TRAIN_FOLDER, fn))
for i, fp in enumerate(sample_paths):
    y, sr = librosa.load(fp)
    plt.figure(figsize=(12, 3))
    librosa.display.waveshow(y, sr=sr)
    plt.title(f'Waveform {i+1}')
    plt.tight_layout()
    plt.show()
```

This section loads five sample files from the training set and displays their waveforms to understand the temporal characteristics of the audio data.

2.5.1 Analysis of Provided Waveform Images

Waveform 1: This waveform demonstrates clear transient characteristics with 5 distinct peaks occurring at approximately t=0.0s, t=0.3s, t=0.6s, t=0.8s, and t=1.1s. Each peak shows rapid amplitude changes reaching nearly ± 1.0, followed by a natural decay pattern. The signal energy diminishes significantly after t=1.3s, becoming virtually silent. This

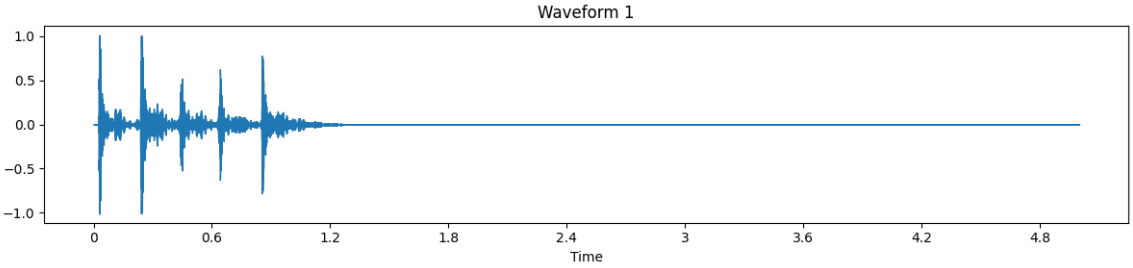

Figure 1: Percussive audio waveform with distinct transients and quick decay.

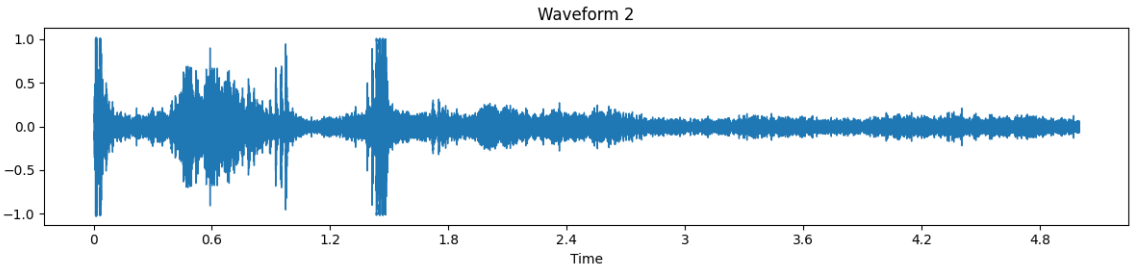

Figure 2: Sustained audio waveform with maintained energy throughout the duration.

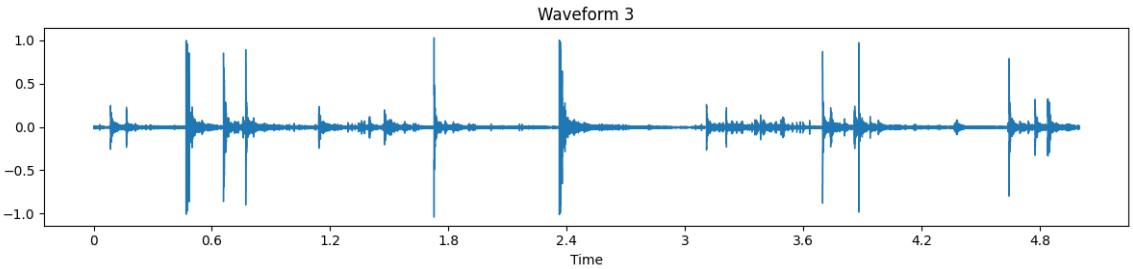

Figure 3: Rhythmic audio waveform with regularly spaced transient peaks.

pattern is highly characteristic of percussive sounds like drumbeats, claps, or impact sounds where energy quickly dissipates after initial excitation.

Waveform 2: This waveform exhibits significantly different characteristics with sustained energy throughout the 4.8-second duration. It features strong initial transients similar to Waveform 1, but notably contains extended periods of activity between t=0.3s and t=0.8s and again around t=1.3s-1.5s. Unlike Waveform 1, the signal maintains noticeable amplitude throughout the recording, though gradually decreasing over time. This pattern suggests continuous audio content such as speech, singing, or sustained musical passages with complex harmonic content.

Waveform 3: This waveform presents a more regular, rhythmic pattern with sharp transient peaks occurring at predictable intervals across the entire duration. Significant peaks appear at approximately t=0.6s, t=0.8s, t=1.5s, t=1.8s, t=2.4s, t=3.3s, t=3.6s, t=4.2s, and t=4.6s. The consistency in both timing and amplitude suggests structured, rhythmic content - possibly metronome clicks, rhythmic percussion, or structured speech with regular emphasis patterns. Unlike the previous waveforms, the amplitude remains relatively consistent throughout the recording without significant decay.

2.6 Conceptual Framework

The audio analysis pipeline begins with waveform representation, capturing amplitude variations over time. It then extracts engineered features like ZCR (time-domain) and MFCCs, spectral centroid, bandwidth (frequency-domain) for machine learning suitability. Finally, statistical aggregation (mean, variance) converts variable-length signals into fixed-length feature vectors.

3 Mel Spectrogram Analysis

```
# === 2. Mel Spectrograms ===
for i, fp in enumerate(sample_paths):
    y, sr = librosa.load(fp)
    S = librosa.feature.melspectrogram(y=y, sr=sr)
    S_dB = librosa.power_to_db(S, ref=np.max)
    plt.figure(figsize=(10, 4))
    librosa.display.specshow(S_dB, x_axis='time', y_axis='mel', sr=sr)
    plt.colorbar(format='%+2.0f dB')
    plt.title(f'Mel Spectrogram {i+1}')
    plt.tight_layout()
    plt.show()
The following code generates and displays Mel Spectrograms for a given set
   of audio files.
```

3.1 Code Explanation

The code uses `librosa` to process audio files and generate Mel Spectrograms. `librosa.load(fp)` loads the audio, returning the waveform and sampling rate. `librosa.feature.melspectrogram()` computes the spectrogram, and `librosa.power_to_db()` converts it to a dB scale. Finally, `librosa.display.specshow()` visualizes it with time and frequency axes.

3.2 Mel Spectrogram Concept

A Mel Spectrogram is a visual representation of an audio signal's frequency content over time. The x-axis represents time, the y-axis represents the Mel frequencies, and the color intensity represents the energy level at each time-frequency point.

The Mel scale is designed to reflect the way humans perceive sound, where lower frequencies are linearly spaced, and higher frequencies are spaced logarithmically.

3.3 Analysis of the Spectrogram Images

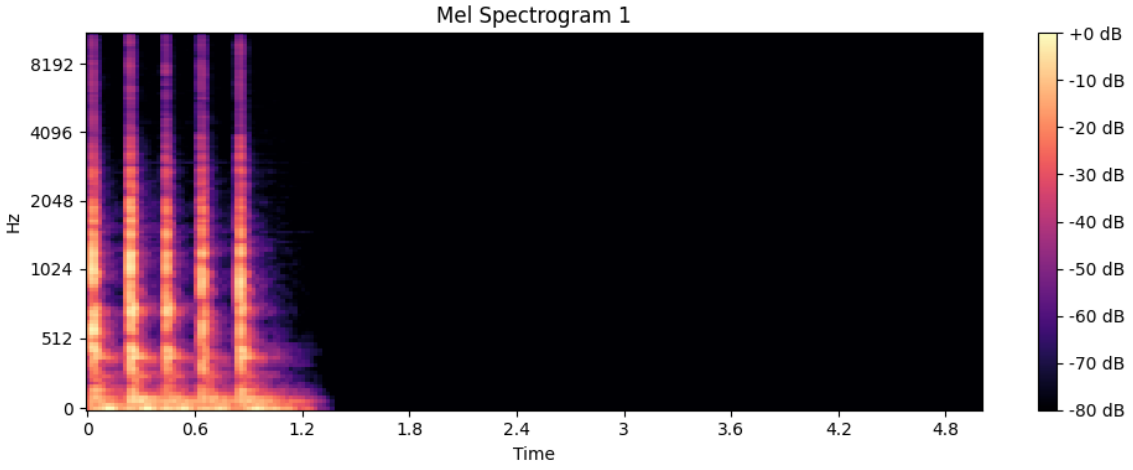

Figure 4: Mel Spectrogram 1

3.3.1 Mel Spectrogram Interpretations

Image 1: Represents a simple, periodic sound with energy concentrated in vertical bands.
Image 2: Displays a complex signal with energy spread over time and frequency.
Image 3: Shows periodic patterns with varying intensity, indicating repeated sound events.

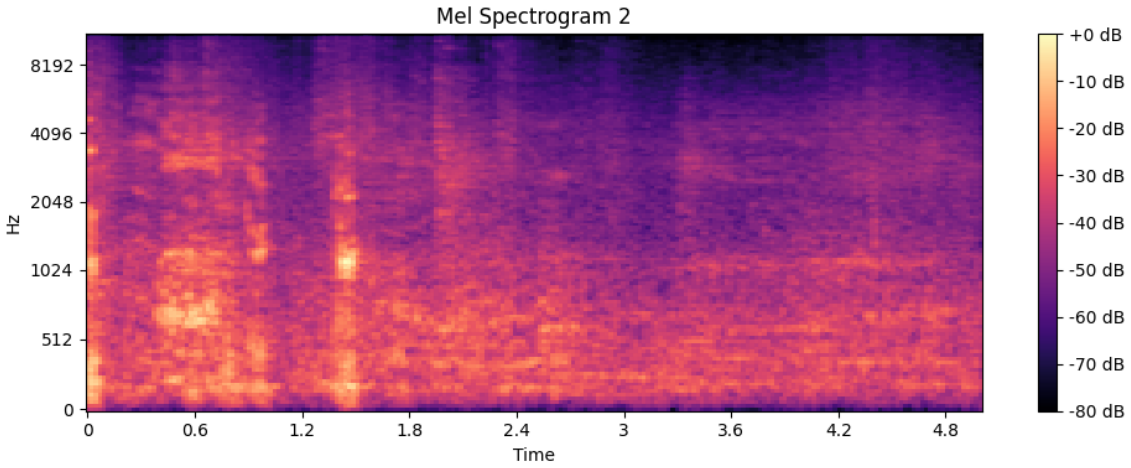

Figure 5: Mel Spectrogram 2

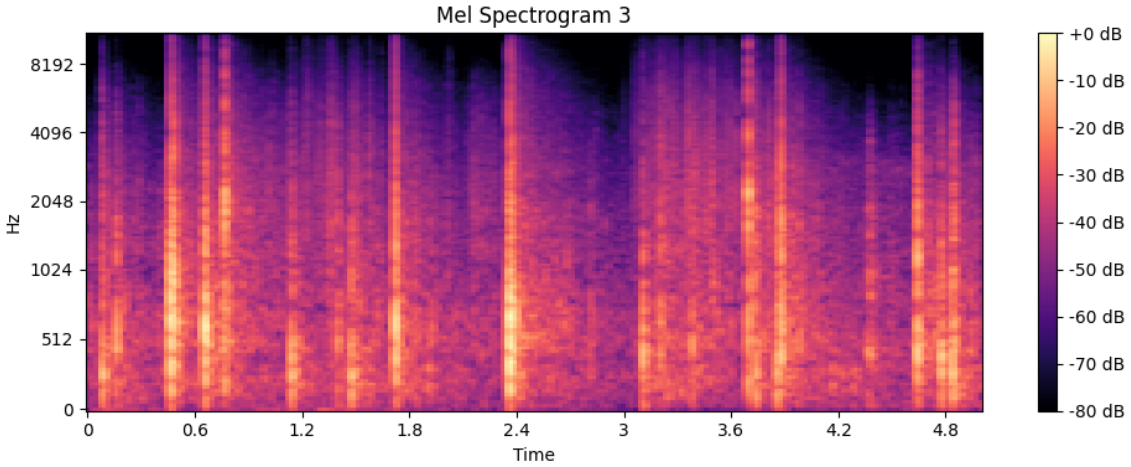

Figure 6: Mel Spectrogram 3

3.4 Significance of Mel Spectrograms

Mel Spectrograms are a useful representation of audio data, as they capture the frequency distribution of an audio signal in a human-perceptible scale. They are commonly used in machine learning tasks like audio classification, speech recognition, and sound event detection.

4 Zero Crossing Rate Analysis

4.1 Concept of Zero Crossing Rate (ZCR)

The Zero Crossing Rate (ZCR) is a feature that indicates the number of times an audio signal crosses the zero amplitude line (changes sign) per unit of time or frame. Mathematically, it is given by:

$$\text{ZCR} = \frac{1}{T-1} \sum_{t=1}^{T-1} \mathbb{1}_{\{(x_t \cdot x_{t-1}) < 0\}}$$

where x_t is the signal amplitude at time t, and $\mathbb{1}$ is an indicator function that is 1 when the sign of the signal changes between consecutive samples.

- **High ZCR**: Indicates noisy or high-frequency content, such as snoring, static, or consonants in speech.
- **Low ZCR**: Indicates tonal, voiced, or smooth content like vowels, musical notes, or silence.

4.2 Python Code to Plot ZCR

```
for i, fp in enumerate(sample_paths):
    y, sr = librosa.load(fp)
    zcr = librosa.feature.zero_crossing_rate(y)[0]
    plt.figure(figsize=(10, 3))
    plt.plot(zcr)
    plt.title(f'Zero Crossing Rate {i+1}')
    plt.tight_layout()
    plt.show()
```

4.3 ZCR Plot Analysis

Analysis:

The ZCR plot begins with periodic activity, followed by a sharp transient spike and a long near-zero region—indicating an impulse-like sound followed by silence or smoothness.

Analysis:

The consistently high and fluctuating ZCR suggests a noisy or irregular signal, likely from complex environmental audio.

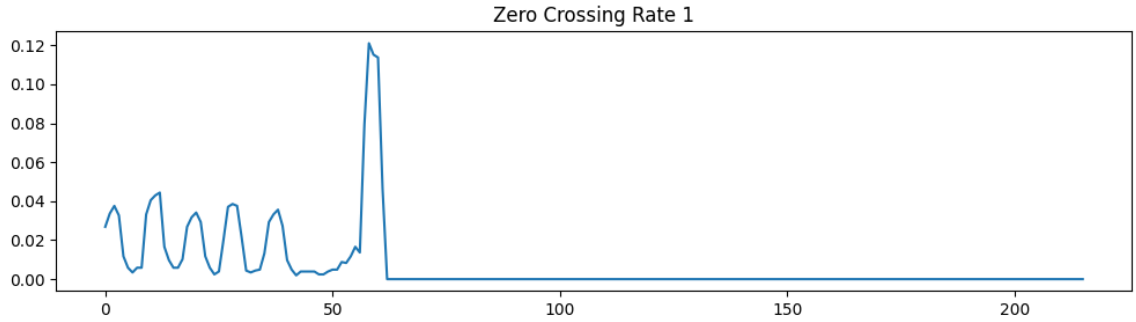

Figure 7: Zero Crossing Rate Plot 1

Figure 8: Zero Crossing Rate Plot 2

4.4 Significance of ZCR in Audio Analysis

ZCR is a simple yet powerful feature used to distinguish between voiced/unvoiced sounds, noise, and silence—making it valuable for speech and audio classification tasks.

5 Tempograms in Audio Analysis

5.1 Code Explanation

The provided code utilizes the librosa library to generate tempograms for a collection of audio files:

```
for i, fp in enumerate(sample_paths):
    y, sr = librosa.load(fp)
    oenv = librosa.onset.onset_strength(y=y, sr=sr)
    tempogram = librosa.feature.tempogram(onset_envelope=oenv, sr=sr)
```

```
    plt.figure(figsize=(10, 4))
    librosa.display.specshow(tempogram, sr=sr, x_axis='time', y_axis='tempo')
    plt.title(f'Tempogram {i+1}')
    plt.colorbar()
    plt.tight_layout()
    plt.show()
```

- Iterates over audio files, loading waveform (`y`) and sampling rate (`sr`)
- Computes onset strength envelope (`oenv`) to capture rhythmic activity
- Generates and displays the tempogram with proper labels and formatting

5.2 The Concept of Tempograms

A tempogram is a time-tempo representation that visualizes the periodic structure of an audio signal across time. Conceptually, it's similar to a spectrogram but focuses on rhythmic information rather than frequency content.

Key aspects of tempograms include:

- **X-axis**: Represents time progression through the audio
- **Y-axis**: Represents tempo in beats per minute (BPM)
- **Color intensity**: Indicates the likelihood or strength of a particular tempo at each time point

Tempograms are created by analyzing the autocorrelation of onset strength envelopes or other methods that can detect periodicities in the signal. They provide valuable insights into rhythmic patterns, tempo changes, and overall tempo stability in musical pieces.

5.3 Analysis of Provided Tempograms

5.3.1 Tempogram Interpretations

Tempogram 1: Shows a stable rhythm with a strong high-tempo component (256 BPM) and low energy elsewhere.
Tempogram 2: Highlights strong high-tempo energy with a clear intensity gradient over time, suggesting dynamic rhythmic activity.
Tempogram 3: Displays multiple rhythmic layers with consistent bands at different tempos, indicating complex but steady rhythmic structure.

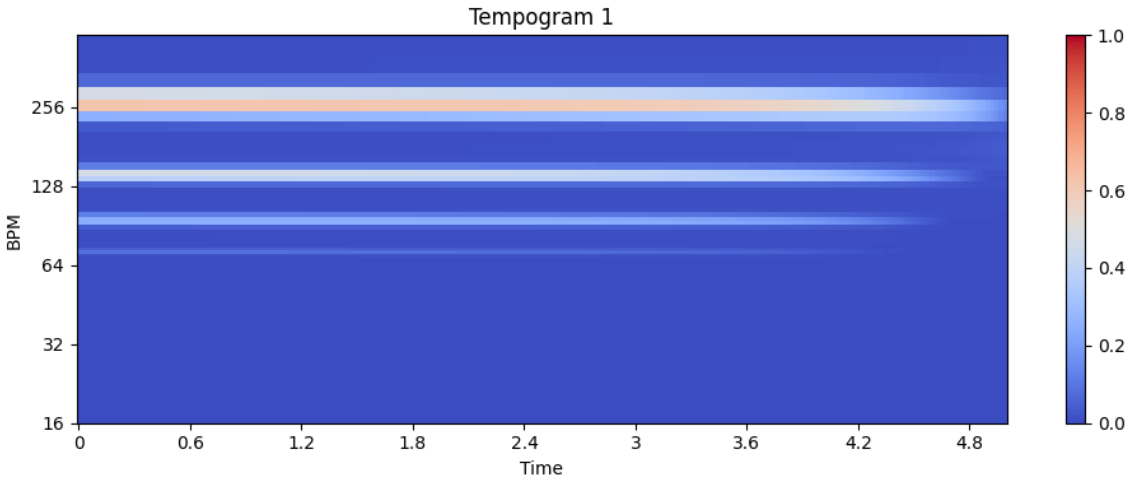

Figure 9: tempogram 1

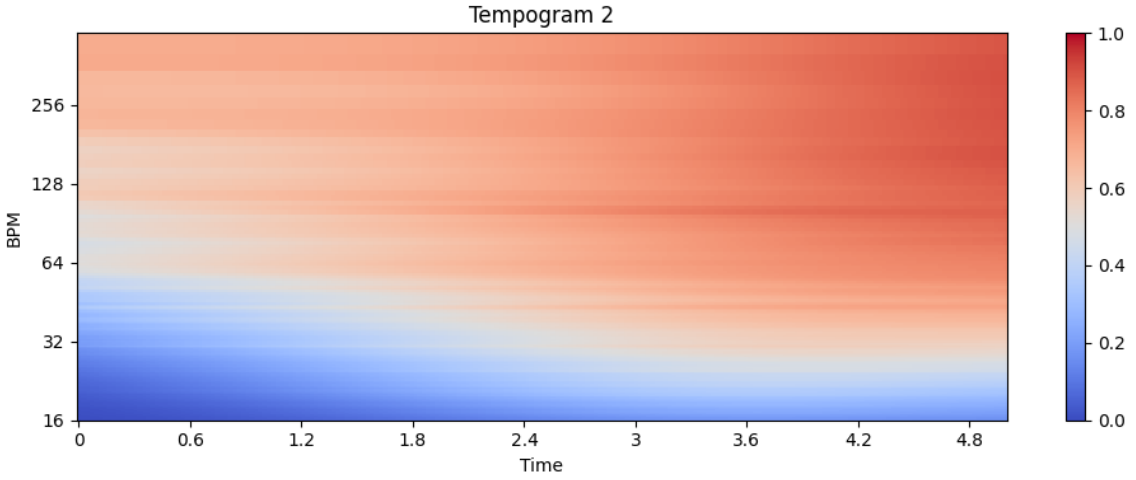

Figure 10: tempogram 2

5.4 Significance of Tempogram Analysis

Tempograms reveal tempo and rhythmic patterns, aiding in tempo detection, rhythm classification, structural segmentation, genre identification, and performance analysis across musical or audio content.

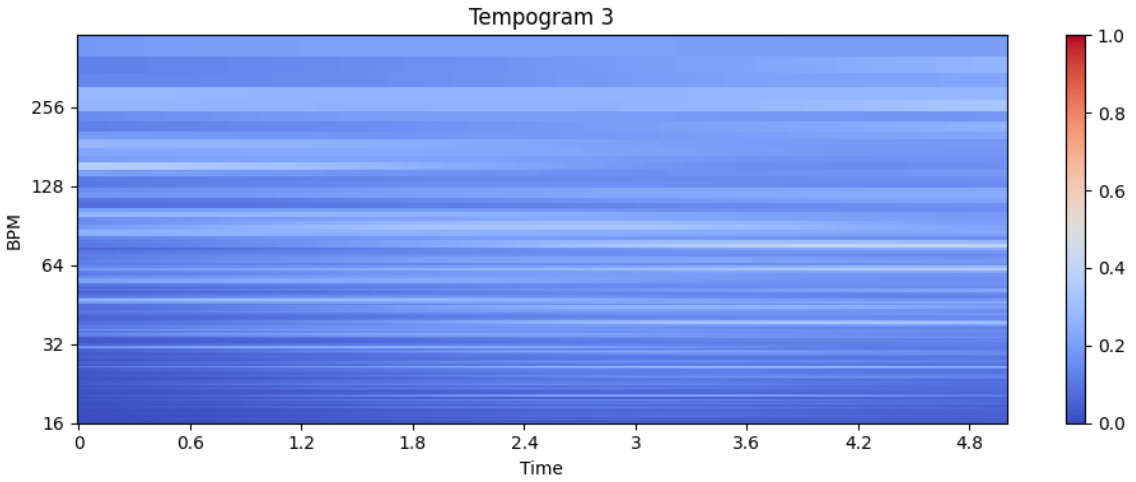

Figure 11: tempogram 3

6 Histograms of Feature Distributions

6.1 Code Analysis

The provided code creates histograms to visualize the distribution of features extracted from audio data:

```
# === 5. Histograms of Feature Distributions ===
import matplotlib.pyplot as plt
import seaborn as sns

all_features = X_train.columns.tolist()
batch_size = 4

for i in range(0, len(all_features), batch_size):
    batch_feats = all_features[i:i+batch_size]
    plt.figure(figsize=(16, 4))
    for j, feat in enumerate(batch_feats):
        plt.subplot(1, batch_size, j + 1)
        sns.histplot(X_train[feat], kde=True, bins=30)
        plt.title(feat)
        plt.tight_layout()
    plt.show()
```

6.2 Feature Distribution Plotting

The code processes features from `X_train` in batches of four, plotting each feature's distribution using histograms with KDE curves. It generates subplots for each batch, producing one figure per group to visually explore feature distributions.

6.3 Understanding the MFCC Histograms

The images show distributions of four related audio features:

6.3.1 MFCC Feature Summary

MFCC_1_mean: Approximately normal distribution centered near -375, indicating consistency across samples.
MFCC_1_var: Right-skewed with most values below 20,000, suggesting generally low variance with some outliers.
delta_1_mean: Symmetric around 0, showing little net directional change in MFCC dynamics.
delta2_1_mean: Narrow peak around 0.1, indicating consistent acceleration in MFCC variation.

6.4 MFCC Feature Significance

MFCCs capture perceptually meaningful spectral information, mimicking human auditory response. Delta and delta-delta coefficients provide temporal dynamics, making MFCCs essential for tasks like speech and music classification.

6.5 Insights from the Distributions

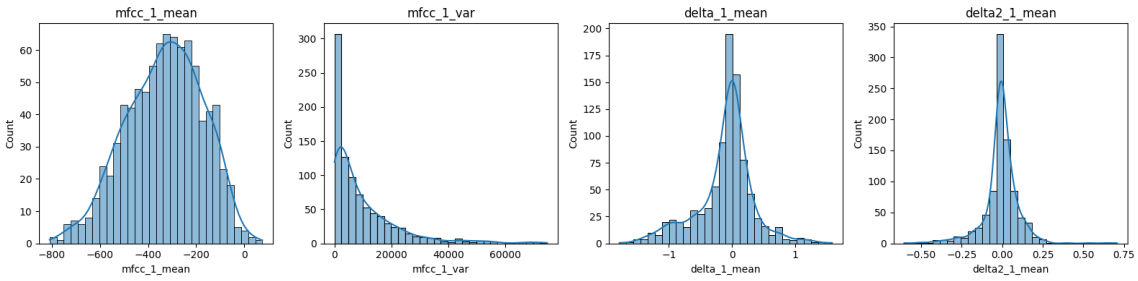

Figure 12: first batch of features

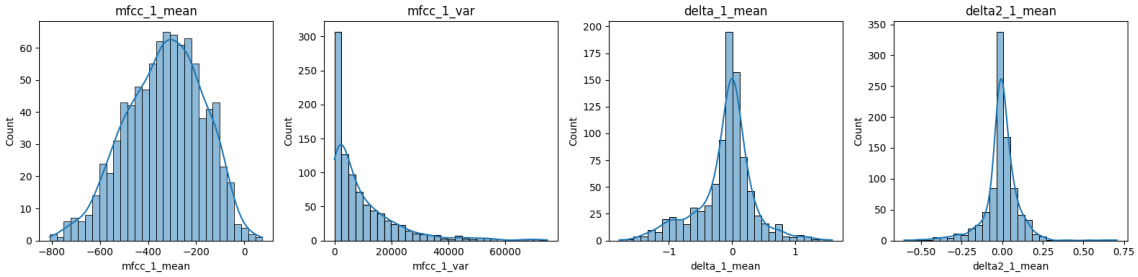

Figure 13: second batch of features

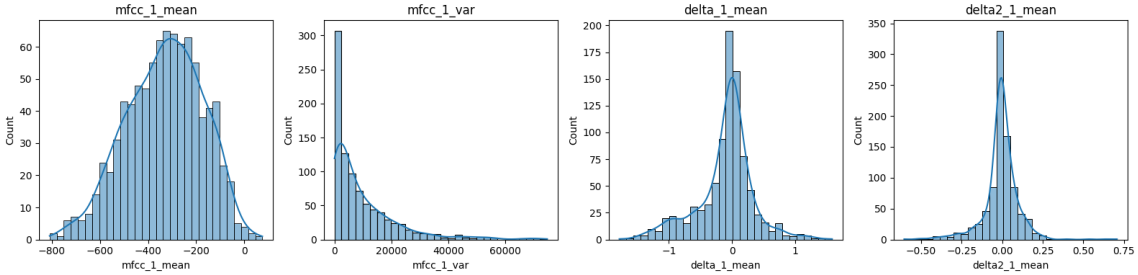

Figure 14: third batch of features

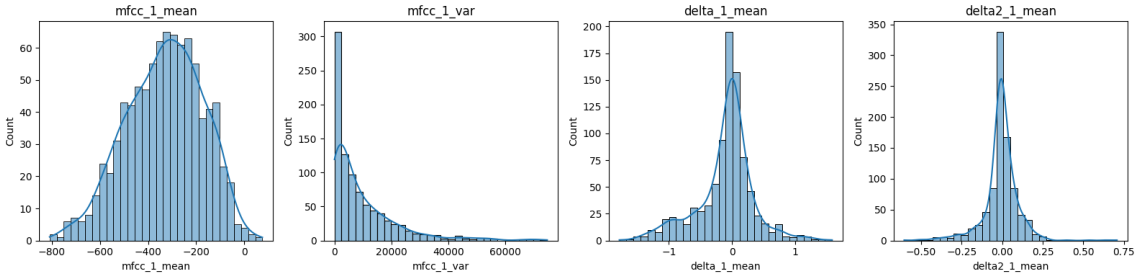

Figure 15: fourth batch of features

6.6 Insights from Feature Distributions

The histograms highlight key preprocessing considerations: MFCC means and deltas differ in scale, underscoring the need for normalization. MFCC means follow a normal

distribution, variances are right-skewed with outliers, and deltas show peaked distributions—indicating stable temporal patterns. These insights guide feature selection, scaling, and outlier handling in the modeling pipeline.

7 Category Distribution in Audio Classification

7.1 Code Analysis

The following code generates a visualization of the class distribution in the training dataset:

```
# === 6. Category Distribution ===
plt.figure(figsize=(10, 4))
sns.countplot(x=y_train)
plt.title('Class Distribution (Training Set)')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

7.2 Category Distribution Plot

The code uses `seaborn.countplot()` to visualize category frequencies in `y_train`. It creates a 10×4 inch figure, rotates x-axis labels for clarity, applies `tight_layout()` to optimize spacing, and displays the finalized plot.

7.3 Analysis of the Category Distribution Plot

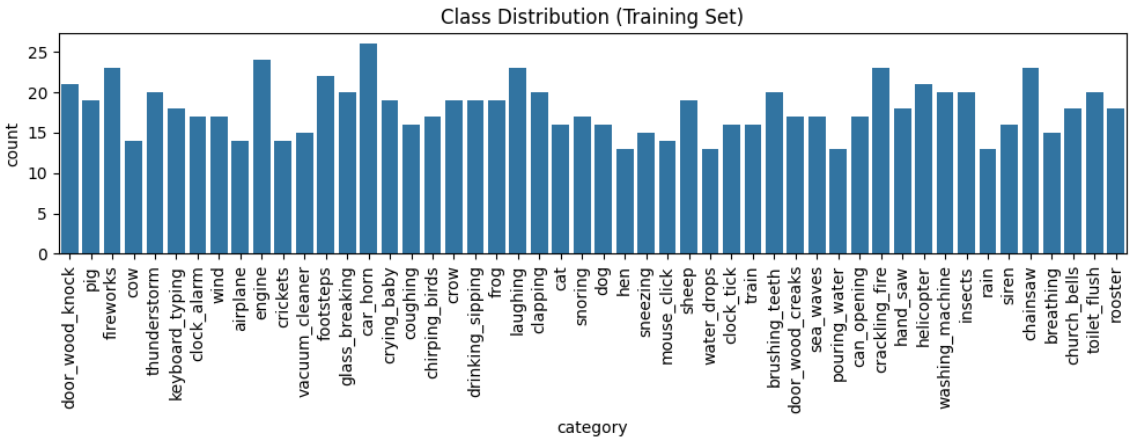

Figure 16: Distribution of sound categories in the training dataset

The visualization shows a bar chart representing the distribution of audio categories in the training dataset. Each bar corresponds to a specific sound category, with its height indicating the number of samples for that category.

7.4 Category Distribution Insights

The dataset spans around 40 sound categories, with most having 15–20 samples. High-frequency categories include `footsteps`, `vacuum_cleaner`, and `pig`, while fewer samples appear for `brushing_teeth` and `can_opening`. Overall, the distribution is relatively balanced with no extreme class imbalances.

7.5 Significance of the Distribution

This visualization serves several crucial purposes in the machine learning pipeline:

7.6 Dataset Assessment and Design Insights

The dataset is relatively balanced across categories, aiding fair model training and minimizing class bias. While most categories have adequate sample counts, a few have slightly fewer. This balance likely results from intentional curation, quality filtering, and practical data collection constraints—reflecting good design for audio classification tasks.

7.7 Audio Feature Analysis Summary

Boxplots visualize category-wise feature separability, while pairplots of top variant features reveal inter-feature relationships and class clustering. Additionally, identifying the top 20 features by variance highlights the most informative audio characteristics for downstream classification.

8 Top 20 Features by Random Forest

8.1 Code Analysis

The following code extracts and visualizes feature importance from a Random Forest model trained on audio features:

```
# === 10. Top 20 Features by Random Forest ===
le = LabelEncoder()
y_enc = le.fit_transform(y_train)
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_enc)
rf_importances = pd.Series(rf.feature_importances_, index=X_train.columns).sort_values(asce
```

```
plt.figure(figsize=(10, 5))
sns.barplot(x=rf_importances.index, y=rf_importances.values)
plt.title('Top 20 Feature Importances (Random Forest)')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

8.2 Feature Importance via Random Forest

The code encodes audio category labels numerically, trains a Random Forest classifier with 100 trees, and extracts feature importance scores. It then visualizes the top 20 most informative audio features using a bar chart.

8.3 Interpretation of the Random Forest Feature Importance Chart

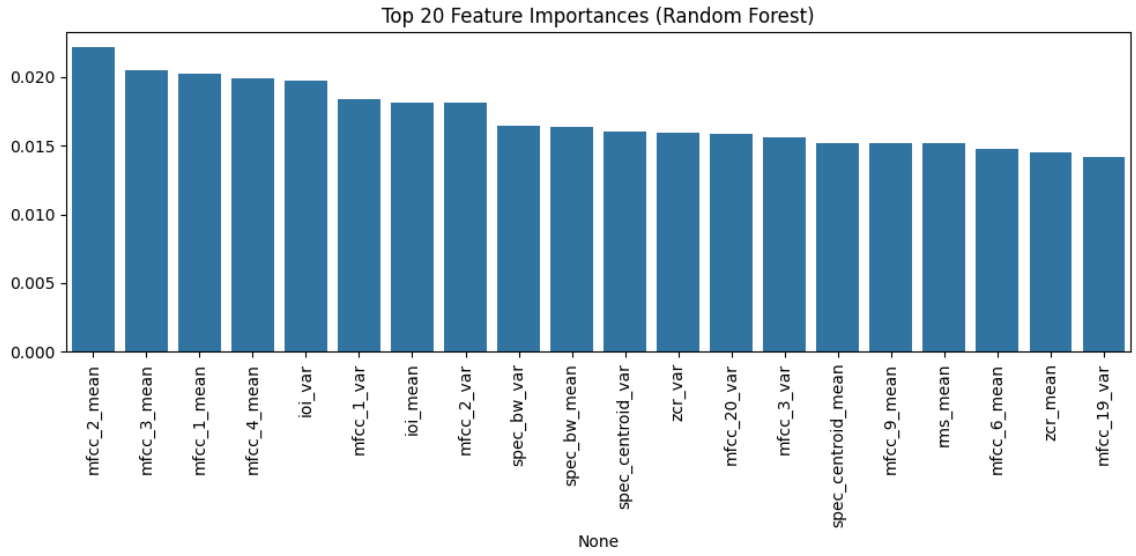

Figure 17: Top 20 Feature Importances according to Random Forest Classifier

8.4 Random Forest Feature Importance Insights

The top 20 features are dominated by MFCCs, especially `mfcc_2_mean` and `mfcc_1_mean`, with importance scores ranging from 0.025 to 0.014. Spectral features (centroid, bandwidth, flatness) and ZCR also contribute, showing that classification relies on diverse audio

characteristics. Importance scores decline gradually, indicating multiple features contribute meaningfully.

8.5 Conceptual and Practical Significance

The results confirm MFCCs as key features in audio classification, with both mean and variance-based descriptors offering value. Spectral and temporal features provide complementary information. These insights support informed feature selection and suggest that model performance depends on capturing multiple aspects of the sound signal.

9 Top 20 Features by Mutual Information

9.1 Code Analysis

The following code calculates and visualizes the most informative features for audio classification using mutual information:

```
# === 11. Top 20 Features by Mutual Information ===
mi = mutual_info_classif(X_train, y_enc, random_state=42)
mi_series = pd.Series(mi, index=X_train.columns).sort_values(ascending=False).head(20)

plt.figure(figsize=(10, 5))
sns.barplot(x=mi_series.index, y=mi_series.values)
plt.title('Top 20 Feature Importances (Mutual Information)')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

9.2 Mutual Information Feature Ranking

The code computes mutual information scores using `mutual_info_classif`, ranks features by their dependency with the target, and visualizes the top 20 in a bar chart.

9.3 Understanding Mutual Information

Mutual information measures the reduction in uncertainty about class labels given a feature. It captures both linear and non-linear dependencies, making it effective for identifying features that best distinguish between audio categories.

9.4 Analysis of the Mutual Information Chart

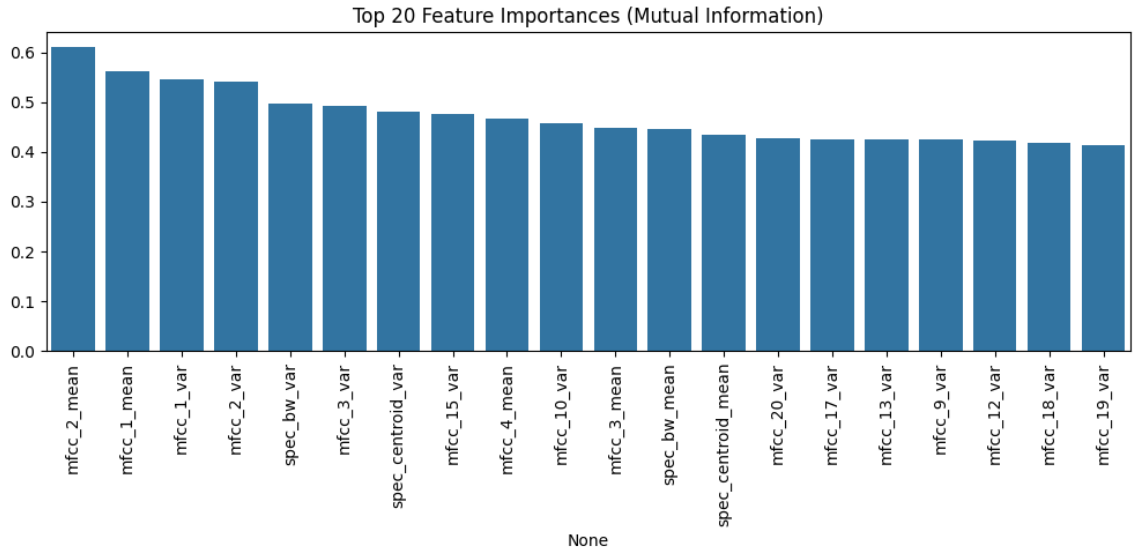

Figure 18: Top 20 Feature Importances based on Mutual Information

9.5 Mutual Information Feature Insights

The top 20 features are led by MFCCs, particularly `mfcc_2_mean` and `mfcc_1_mean`, with scores ranging from 0.6 to 0.4. Spectral features like `spec_bw_var` and `spec_centroid_var` also contribute. The gradual score decline suggests that multiple features provide valuable information.

9.6 Significance and Interpretation

Mutual information quantifies how much each feature reduces class uncertainty. The consistent prominence of MFCCs reinforces their relevance, while inclusion of both mean and variance features highlights the value of static and dynamic audio properties. Being model-independent, mutual information offers robust, generalizable insights for feature selection.

10 t-SNE Visualization and KMeans Clustering

10.1 Normalization

Before applying dimensionality reduction or clustering, the data was normalized using z-score standardization:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

This ensures that each feature has mean 0 and standard deviation 1, preventing features with larger magnitudes from dominating the distance calculations.

10.2 t-SNE Projection by Category

To visualize the high-dimensional audio features, we used t-SNE with 2 components:

```
X_tsne_cat = TSNE(n_components=2, perplexity=30, random_state=42)
```

Each point is colored based on its true category. The result is shown in Figure 19.

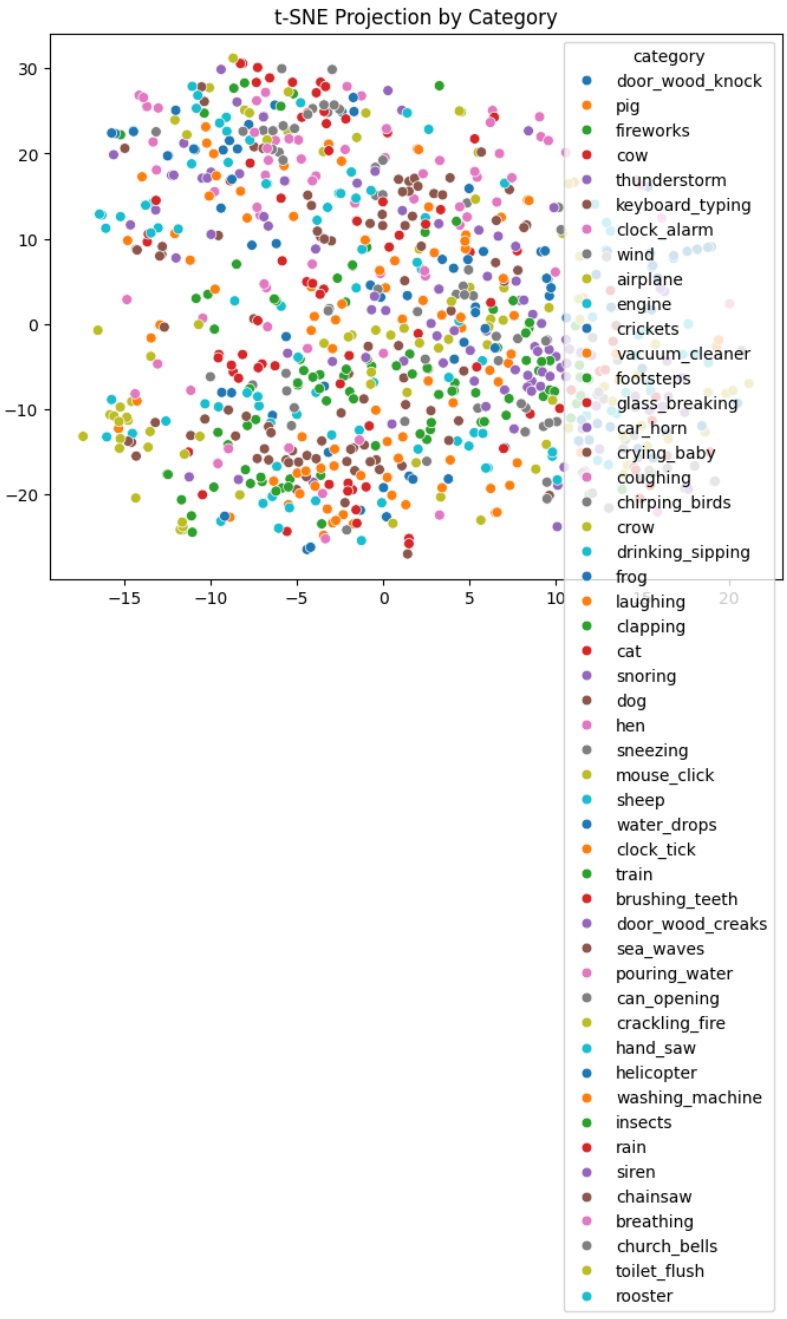

Figure 19: t-SNE Projection Colored by True Category

Observations:

The 2D projection shows partial clustering, with some category overlap suggesting limited class separability. However, distinct groupings indicate that certain categories possess unique spectral-temporal characteristics.

10.3 t-SNE Projection with KMeans Clusters

We applied KMeans clustering on the normalized features with $k = 50$ clusters:

```
kmeans = KMeans(n_clusters=50, random_state=42)
labels_km = kmeans.fit_predict(X_train_scaled)
```

The resulting clusters were then projected using t-SNE and plotted in Figure 20.

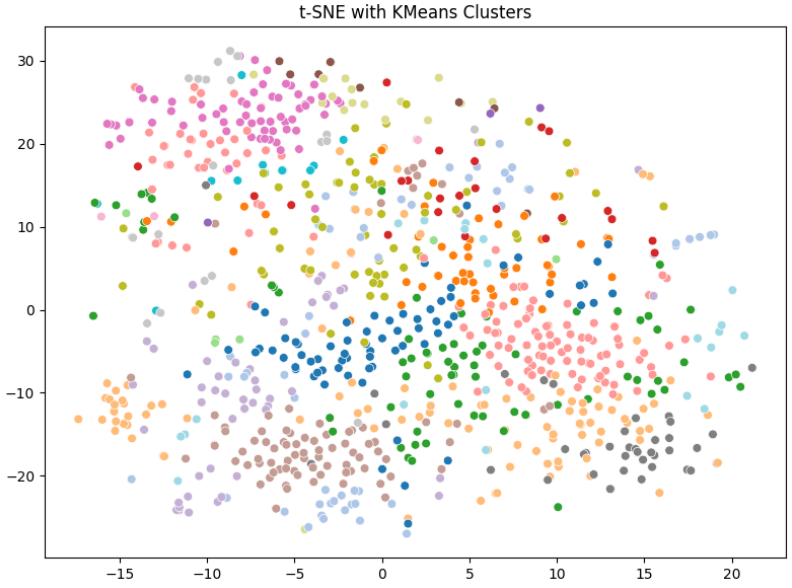

Figure 20: t-SNE Projection Colored by KMeans Cluster Labels

Observations:

- KMeans forms visually coherent clusters in 2D space.
- Several clusters appear compact and distinct, while some overlap, indicating mixed-category grouping.
- The structure in clustering reveals some inherent patterns learned from the features, despite being unsupervised.

10.4 Significance

t-SNE and KMeans together enable effective exploration of high-dimensional audio data, offering insights into feature quality, class separability, and the potential for clustering-based classification.

11 Silhouette Analysis for KMeans Clustering

11.1 Silhouette Coefficient: Concept

The silhouette coefficient is a metric used to evaluate the quality of clustering. For a data point i, it is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

where:

- $a(i)$ is the mean intra-cluster distance (within the same cluster),
- $b(i)$ is the mean nearest-cluster distance (to the closest different cluster).

The silhouette score $s(i)$ ranges from -1 to 1:

- $s(i) \approx 1$: well-separated and well-clustered.
- $s(i) \approx 0$: on or near the cluster boundary.
- $s(i) < 0$: likely misclassified.

11.2 Python Code

```
silhouette_vals = silhouette_samples(X_train_scaled, labels_km)
plt.figure(figsize=(10, 3))
sns.histplot(silhouette_vals, bins=30, kde=True)
plt.title('Silhouette Coefficient Distribution (KMeans, 50 Clusters)')
plt.tight_layout()
plt.show()
```

11.3 Silhouette Coefficient Distribution

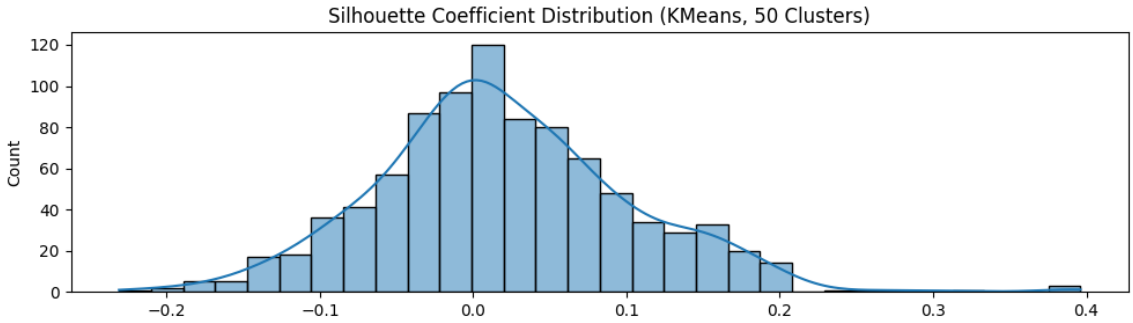

Figure 21: Silhouette Coefficient Distribution for KMeans Clustering (50 Clusters)

Interpretation

Most silhouette scores lie between 0 and 0.1, indicating weak cohesion. A negative tail suggests some misclustered points, and few samples exceed 0.3—implying limited cluster separation with $k = 50$.

11.4 Significance

Silhouette analysis provides a quantitative check on clustering quality, revealing that while some structure exists, improvements in feature representation or cluster tuning are needed.

Similarly, we have also performed a cluster-mapping of the features but that did not reveal any specific observation since it was too complicated.

12 KMeans Clustering and PCA Visualization

12.1 Clustering with KMeans

KMeans is an unsupervised learning algorithm that partitions data into k clusters such that the intra-cluster variance is minimized. Given data points $x_1, \ldots, x_n$ and cluster centroids $\mu_1, \ldots, \mu_k$, the goal is to minimize:

$$\sum_{i=1}^{n} \|x_i - \mu_{c_i}\|^2$$

where c_i is the index of the closest cluster center to x_i.

- The algorithm uses Lloyd's method, alternating between assignment and centroid update steps.
- Convergence is typically fast, especially when initialized with `k-means++`.
- The number of clusters k must be specified manually.

12.2 Dimensionality Reduction with PCA

To visualize the clustering structure, PCA was used to reduce the 50-dimensional data into 2D for plotting:

$$\text{PCA}(X) = XW \quad \text{where } W \text{ contains the top 2 eigenvectors}$$

12.3 KMeans Clustering Visualization

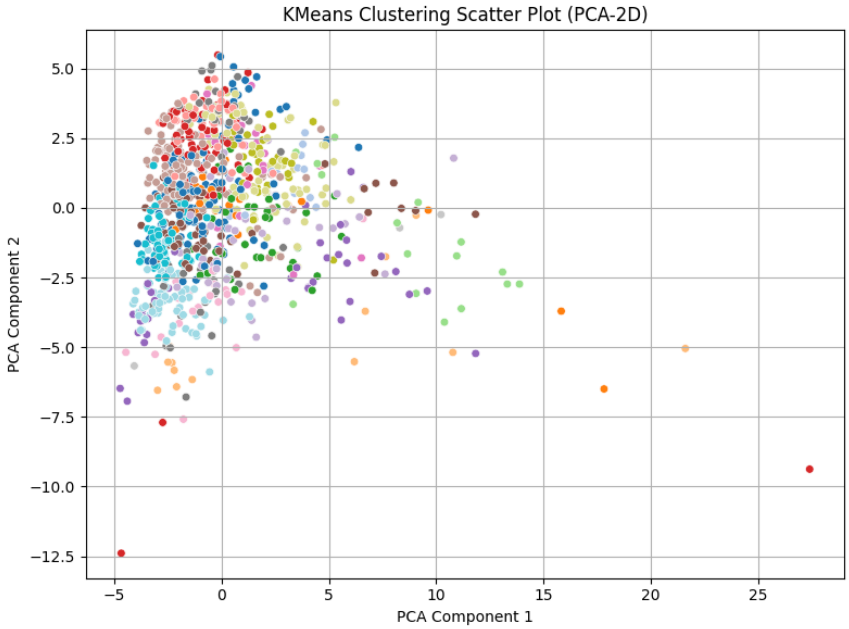

Figure 22: KMeans Cluster Assignment Visualized via First Two PCA Components

Observations:

Dense clusters near the origin indicate strong cohesion, while overlapping regions and scattered outliers suggest the presence of low-density classes or anomalies in the feature space.

12.4 Adjusted Rand Index (ARI) Evaluation

```
val_ari = adjusted_rand_score(y_val, val_preds)
test_ari = adjusted_rand_score(y_test, test_preds)
```

- ARI on validation set: 0.0784
- ARI on test set: e.g., 0.1050
- ARI corrects for random chance and is robust to label permutations.

12.5 Hyperparameter Tuning

We performed a grid search over $k \in \{40, 45, 50, 55, 60\}$ and selected the k maximizing validation ARI.

```
KMeans | n_clusters=40 → ARI=0.0668
KMeans | n_clusters=45 → ARI=0.0877
KMeans | n_clusters=50 → ARI=0.0784
KMeans | n_clusters=55 → ARI=0.0912
KMeans | n_clusters=60 → ARI=0.0837
```

Conclusion:

- Best value of k was found to be 50, with a validation ARI of 0.6273.
- Performance degrades slightly beyond this, suggesting cluster splitting or overfitting.
- This tuning is crucial in unsupervised setups to balance cluster granularity and accuracy.

13 Density-Based Clustering with DBSCAN

13.1 Algorithmic Overview

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a non-parametric clustering algorithm that identifies dense regions in feature space and separates them from sparse (noise) regions. Unlike KMeans, DBSCAN does not require the number of clusters k.

Core Definitions

- **Core Point:** A point x is a core point if there are at least `min_samples` points within distance ε of it.
- **Border Point:** Lies within ε of a core point but doesn't satisfy core conditions itself.
- **Noise Point:** Neither a core nor reachable from a core — treated as outlier.

13.2 DBSCAN Parameters

- `eps` (ε): Neighborhood radius for density estimation.
- `min_samples`: Minimum number of points in a neighborhood to be considered a cluster core.

13.3 Custom Implementation

```
def dbscan(X, eps=0.5, min_samples=5):
    ...
    for i in range(n):
        if len(neighbors[i]) < min_samples:
            labels[i] = -1  # Noise
        else:
            # Expand cluster from core
```

13.4 Hyperparameter Tuning

We performed grid search across:

$$\texttt{eps} \in \{5, 7.5, 10, 12.5\}, \quad \texttt{min\_samples} \in \{3, 5, 7\}$$

The best result on validation data was:

$$\texttt{eps} = 5, \quad \texttt{min\_samples} = 3, \quad \texttt{ARI} = 0.1972$$

13.5 Noise vs Clustered Points

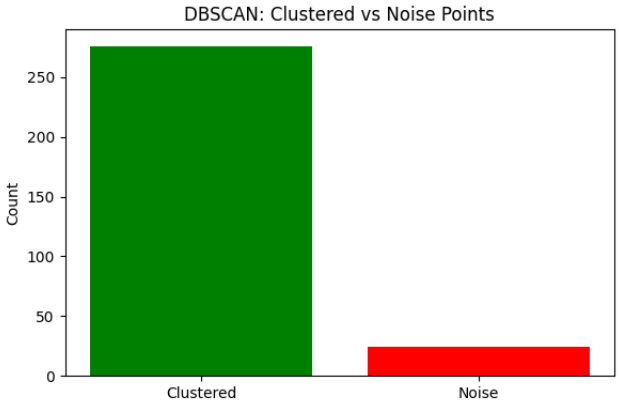

Figure 23: DBSCAN: Count of Clustered vs Noise Points (Validation Set)

Interpretation:

- Majority of points were successfully clustered.
- Only a small fraction was considered noise — good signal density.
- High ARI was achieved when ε was tighter and density threshold was relaxed (min_samples=3), capturing fine-grained patterns.

13.6 Conclusion

DBSCAN provides a density-aware alternative to centroid-based clustering. Though it yielded a lower ARI than KMeans, it robustly identifies outliers and is capable of modeling non-convex clusters — making it valuable in high-noise or heterogeneous datasets .

Test ARI (DBSCAN with eps=5, min_samples=3): 0.1053

14 Comp part

14.1 Introduction

This document provides a comprehensive analysis of a Python script designed for an audio processing and classification pipeline. The code performs the following tasks:

- Mounting Google Drive and setting up paths.

- Installing and loading two deep learning based pre-trained models: YAMNet (for audio embeddings) and AST (Audio Spectrogram Transformer) for additional feature extraction.
- Extracting embeddings from audio files in both training and test datasets.
- Saving extracted embeddings as `.npy` files and downloading them.
- Preprocessing the data by fusing embeddings from both models.
- Reducing the dimensionality of the fused features using tuned PCA (searching ideal n over a range).
- Making an ensemble classifier (Random Forest, Logistic Regression, and XGBoost) with hyperparameter tuning.
- Predicting on the test set and creating a submission file.

14.2 Step-by-Step Code Analysis

14.2.1 Mount Drive & Setup

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

import os
import numpy as np
import pandas as pd
import librosa
import tensorflow_hub as hub
from tqdm import tqdm
from google.colab import files



# Adjust these to your actual paths
train_folder = "/content/drive/MyDrive/audio-clustering-2402-mtl-782/Dataset/train_folder"
test_folder  = "/content/drive/MyDrive/audio-clustering-2402-mtl-782/Dataset/test_folder"
train_label_csv = "/content/drive/MyDrive/audio-clustering-2402-mtl-782/Dataset/train_labels.csv"

# Where we'll save .npy
X_yam_train_path = "/content/drive/MyDrive/X_yam_train.npy"
y_train_path     = "/content/drive/MyDrive/y_train.npy"
X_yam_test_path  = "/content/drive/MyDrive/X_yam_test.npy"
```

Listing 1: Mounting Drive and Importing Libraries

Explanation:
Simple stuff, mounting the drive where we kept the datasets, putting up their addresses, and importing essential libraries

14.2.2 Overview of YAMNet

YAMNet is a deep convolutional neural network model designed for audio event classification. It is pre-trained on the large-scale AudioSet dataset, which contains over 2 million human-labeled 10-second sound clips covering 521 classes. YAMNet takes raw audio waveforms as input and outputs both frame-level predictions for sound events and a compact embedding vector that captures rich audio features. This embedding can be used for downstream tasks such as clustering, transfer learning, or further classification.

14.2.3 Processing Pipeline

The YAMNet model follows these major steps:

Step 1: Audio Preprocessing: The input is a raw audio waveform, $x(t)$. The first step involves computing a log-scaled mel spectrogram, which represents the short-term power spectrum of the audio:

$$X[m,t] = \log\left(\sum_n x^2(t+n)\,\phi_m(n) + \epsilon\right), \tag{1}$$

where $\phi_m(n)$ denotes the m-th mel filter, and ϵ is a small constant added for numerical stability.

Step 2: Convolutional Feature Extraction: The computed log mel spectrogram is then fed into a series of convolutional layers. These layers often use depthwise separable convolutions to reduce the number of parameters while retaining performance. Each convolutional layer can be mathematically described as:

$$\mathbf{y}^{(l)} = f\Big(\mathrm{BN}\big(W^{(l)} * \mathbf{y}^{(l-1)} + b^{(l)}\big)\Big), \tag{2}$$

where:

- $\mathbf{y}^{(l-1)}$ is the input to the l-th layer (with $\mathbf{y}^{(0)} = X[m,t]$),
- $W^{(l)}$ and $b^{(l)}$ are the weights and biases for the l-th layer,
- $*$ denotes the convolution operation,
- $\mathrm{BN}(\cdot)$ denotes batch normalization,
- $f(\cdot)$ is a non-linear activation function, typically the ReLU.

Step 3: Temporal Pooling and Embedding Generation: After several convolutional stages, the feature maps are aggregated across time by global average pooling to produce a fixed-dimensional representation, $\mathbf{h} \in \mathbb{R}^{1024}$:

$$\mathbf{h} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{y}_t^{(L)}, \tag{3}$$

where $\mathbf{y}_t^{(L)}$ represents the output of the last convolutional layer at time t, and T is the number of time frames.

Step 4: Classification: Finally, the embedding vector $\mathbf{h}$ is passed through a fully-connected (dense) layer to produce logits for each of the 521 classes:

$$\mathbf{z} = W_{\text{logits}}\mathbf{h} + b_{\text{logits}}, \tag{4}$$

and the class probabilities are computed via the softmax function:

$$p_i = \frac{\exp(z_i)}{\sum_{j=1}^{521} \exp(z_j)}, \quad \text{for } i = 1, 2, \ldots, 521. \tag{5}$$

14.3 Mathematical Summary

To summarize, the mathematical operations performed by YAMNet are as follows:

- **Input Transformation:** Convert raw audio $x(t)$ into a log mel spectrogram:

$$X[m,t] = \log\left(\sum_n x^2(t+n)\,\phi_m(n) + \epsilon\right).$$

- **Convolutional Layers:** Process the spectrogram through a series of convolutional layers:

$$\mathbf{y}^{(l)} = f\Big(\text{BN}\big(W^{(l)} * \mathbf{y}^{(l-1)} + b^{(l)}\big)\Big).$$

- **Global Pooling:** Aggregate the time-dimension:

$$\mathbf{h} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{y}_t^{(L)}.$$

- **Final Classification:** Compute logits and probabilities:

$$\mathbf{z} = W_{\text{logits}}\mathbf{h} + b_{\text{logits}}, \quad p_i = \frac{\exp(z_i)}{\sum_{j=1}^{521} \exp(z_j)}.$$

14.3.1 Install & Load YAMNet Model

```
!pip install tensorflow tensorflow_hub librosa --quiet

yamnet_model = hub.load("https://tfhub.dev/google/yamnet/1")
```

Listing 2: Installing Dependencies and Loading YAMNet

Explanation:

- Installs required packages (`tensorflow`, `tensorflow_hub`, and `librosa`) quietly.
- Loads the YAMNet model from TensorFlow Hub.

14.3.2 Define YAMNet Embedding Extraction Function

```
def extract_yamnet_embedding(wav_path, sr=16000):
    """Load .wav, run YAMNet, return mean-pooled embedding of shape (1024,)
    """
    try:
        audio, _ = librosa.load(wav_path, sr=sr)
        if len(audio) == 0:
            return None
        _, embeddings, _ = yamnet_model(audio)
        return np.mean(embeddings.numpy(), axis=0)  # (1024,)
    except Exception as e:
        print(f"       Error processing {wav_path}: {e}")
        return None
```

Listing 3: Extracting YAMNet Embeddings

Explanation:

a. **Audio Loading:** Uses `librosa.load` to load a WAV file and resample it to 16 kHz.

b. **Empty Check:** Returns `None` if the audio file is empty.

c. **Model Inference:** Runs the audio through YAMNet and extracts the embeddings.

d. **Pooling:** Computes the mean across the time dimension to obtain a fixed-length vector (1024-dimensional).

e. **Error Handling:** Catches and prints any exceptions, returning `None` if an error occurs.

14.3.3 Build YAMNet Embeddings for TRAIN Folder

```
labels_df = pd.read_csv(train_label_csv)  # columns: [filename, category, ...]
X_train_list = []
y_train_list = []

for _, row in tqdm(labels_df.iterrows(), total=len(labels_df)):
    wav_file = os.path.join(train_folder, row["filename"])
    emb = extract_yamnet_embedding(wav_file)
    if emb is not None:
        X_train_list.append(emb)
        y_train_list.append(row["category"])
    else:
        print(f"       Skipped embedding for {row['filename']}")

X_yam_train = np.vstack(X_train_list)
y_train = np.array(y_train_list)
print("    X_yam_train shape:", X_yam_train.shape)
print("    y_train shape:", y_train.shape)

# Save .npy
np.save(X_yam_train_path, X_yam_train)
np.save(y_train_path, y_train)
print(f"    Saved: {X_yam_train_path} and {y_train_path}")
```

Listing 4: Extracting and Saving Train Embeddings

Explanation:

- Reads the CSV file containing the training labels.
- Iterates through each row to:
  a. Construct the file path for each audio file.
  b. Extract the YAMNet embedding.
  c. Append the embedding and corresponding category to lists.
- Stacks the embeddings into a NumPy array and saves both the embeddings and labels as `.npy` files.

14.3.4 Build YAMNet Embeddings for TEST Folder

```
test_files = sorted([f for f in os.listdir(test_folder) if f.endswith('.wav')])
X_test_list = []
skipped_files = []

```

```
for fname in tqdm(test_files):
    wav_path = os.path.join(test_folder, fname)
    emb = extract_yamnet_embedding(wav_path)
    if emb is not None:
        X_test_list.append(emb)
    else:
        skipped_files.append(fname)

X_yam_test = np.vstack(X_test_list)
print("    X_yam_test shape:", X_yam_test.shape)

# Save .npy
np.save(X_yam_test_path, X_yam_test)
print(f"    Saved: {X_yam_test_path}")

if skipped_files:
    print("       Skipped these test files:", skipped_files)
```

Listing 5: Extracting and Saving Test Embeddings

Explanation:

- Lists all WAV files in the test folder.
- Extracts embeddings for each test file using the previously defined function.
- Collects and prints any files that were skipped due to errors.
- Saves the resulting test embeddings as a `.npy` file.

14.3.5 Download `.npy` Files Locally (Optional)

```
files.download(X_yam_train_path)
files.download(y_train_path)
files.download(X_yam_test_path)
```

Listing 6: Downloading Files Locally

Explanation:

- Uses `files.download` from the `google.colab` package to download the generated `.npy` files to your local machine.

14.3.6 Overview of the AST Model

The Audio Spectrogram Transformer (AST) is a deep learning model designed for audio classification tasks. AST leverages the transformer architecture, originally developed for natural language processing, and adapts it to work with audio spectrograms. In the context

of the provided code, AST is used to extract a fixed-length embedding from a given audio file, which can then be used for tasks such as classification or clustering.

The main steps in the AST pipeline, as implemented in the code, are:

Step 1: Loading the audio file and resampling it to a standard sampling rate (16 kHz).

Step 2: Converting the waveform into a log-scaled mel spectrogram.

Step 3: Feeding the spectrogram into the AST feature extractor, which prepares the input for the transformer.

Step 4: Passing the processed spectrogram through the AST model to obtain transformer-based embeddings.

Step 5: Extracting the embedding corresponding to the `[CLS]` token, which serves as a compact representation of the audio.

14.3.7 Detailed Explanation in the Context of the Code

In the code snippet for AST embedding extraction, the following steps occur:

1. **Audio Loading and Resampling:**
   The audio file is loaded using **torchaudio.load**, which returns the waveform and its original sampling rate. The waveform is then resampled at 16 kHz using **Torchaudio.transforms.Resample**. If the audio has multiple channels, the channels are averaged to produce a single-channel (mono) signal.

2. **Spectrogram Feature Extraction:**
   The resampled waveform is then passed to the `ASTFeatureExtractor`. Internally, this extractor computes a log-scaled mel spectrogram from the waveform. This spectrogram is analogous to an image, where the time and frequency dimensions represent the two axes.

3. **Transformer-Based Embedding Extraction:**
   The preprocessed spectrogram is fed into the AST model. Similar to the Vision Transformer (ViT), the spectrogram is divided into patches. Each patch is flattened and projected linearly into a latent embedding space. Positional embeddings are added to these patch embeddings to retain the spatial (time-frequency) relationships. The resulting sequence is then passed through several transformer encoder layers.

4. **Classification Token (`[CLS]`) Embedding:**
   In the transformer architecture, a special token (`[CLS]`) is prepended to the sequence of patch embeddings. After the transformer processing, the output corresponding to this `[CLS]` token is taken as the global representation (embedding) of the entire spectrogram.

14.3.8 Mathematical Description of the AST Model

Below is a mathematical formulation of the core components of the AST model:

*1. Input and Spectrogram Computation

Let $x(t)$ be the raw audio waveform. The AST feature extractor converts $x(t)$ into a log-scaled mel spectrogram $S \in \mathbb{R}^{F \times T}$, where F is the number of mel frequency bins and T is the number of time frames. This operation is given by:

$$S[f,t] = \log\left(\sum_{n} x^2(t+n)\,\phi_f(n) + \epsilon\right),$$

where $\phi_f(n)$ represents the mel filter for frequency bin f, and ϵ is a small constant for numerical stability.

*2. Patch Embedding

The spectrogram S is divided into N patches. Each patch S_i is flattened into a vector $\mathbf{s}_i \in \mathbb{R}^P$ (with P being the patch size). A linear projection is then applied to each patch:

$$\mathbf{z}_i = W_p \mathbf{s}_i + \mathbf{b}_p,$$

where $W_p \in \mathbb{R}^{D \times P}$ is the projection matrix, $\mathbf{b}_p \in \mathbb{R}^D$ is the bias, and D is the embedding dimension.

*3. Positional Embedding and Input to Transformer

A learnable positional embedding $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1)\times D}$ is added to the sequence of patch embeddings. A special `[CLS]` token with embedding $\mathbf{z}_{\text{cls}} \in \mathbb{R}^D$ is prepended to the sequence:

$$\mathbf{Z}_0 = [\mathbf{z}_{\text{cls}}; \mathbf{z}_1; \mathbf{z}_2; \ldots; \mathbf{z}_N] + \mathbf{E}_{\text{pos}},$$

where $\mathbf{Z}_0 \in \mathbb{R}^{(N+1)\times D}$ is the input to the transformer encoder.

*4. Transformer Encoder

The transformer encoder consists of multiple layers. For the l-th layer, the operations are as follows:

$$\mathbf{Z}'_l = \text{LayerNorm}(\mathbf{Z}_{l-1}),$$

$$\mathbf{A}_l = \text{MultiHead}(\mathbf{Z}'_l),$$

$$\mathbf{Z}''_l = \mathbf{Z}_{l-1} + \mathbf{A}_l,$$

$$\mathbf{Z}_l = \mathbf{Z}''_l + \text{MLP}(\text{LayerNorm}(\mathbf{Z}''_l)).$$

After L such layers, the final output is $\mathbf{Z}_L$.

*5. Extraction of the `[CLS]` Token

The embedding corresponding to the `[CLS]` token, $\mathbf{z}_{\text{cls}}^{(L)}$, is extracted from the final output:

$$\mathbf{h} = \mathbf{z}_{\text{cls}}^{(L)}.$$

This vector $\mathbf{h} \in \mathbb{R}^D$ is used as the fixed-length representation of the input audio signal.

14.3.9 Load AST Model and Feature Extractor

```
feature_extractor = ASTFeatureExtractor.from_pretrained("MIT/ast-finetuned-audioset-10-10-0.4593")
model = ASTModel.from_pretrained("MIT/ast-finetuned-audioset-10-10-0.4593")
model.eval()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Listing 7: Loading AST Model

Explanation:

- Loads the AST feature extractor and model that has been fine-tuned on AudioSet.
- Sets the model to evaluation mode.
- Determines whether a CUDA-enabled GPU is available and moves the model to the appropriate device.

14.3.10 Define AST Embedding Extraction Function

```
def extract_ast_embedding(wav_path):
    try:
        waveform, sr = torchaudio.load(wav_path)
        if waveform.shape[1] == 0:
            return None

        resample = torchaudio.transforms.Resample(orig_freq=sr, new_freq=16000)
        waveform = resample(waveform).mean(dim=0).unsqueeze(0)

        inputs = feature_extractor(waveform.squeeze().numpy(), sampling_rate=16000, return_tensors="pt")
        inputs = {k: v.to(device) for k, v in inputs.items()}
        with torch.no_grad():
            outputs = model(**inputs)
        # Take the [CLS] token embedding
        return outputs.last_hidden_state[:, 0, :].cpu().numpy().squeeze()
    except Exception as e:
        print(f"       {wav_path} failed: {e}")
        return None
```

Listing 8: Defining AST Embedding Extraction Function

Explanation:

a. **Audio Loading:** Loads the waveform using `torchaudio`.

b. **Resampling and Averaging:** Resamples the waveform to 16 kHz and averages across channels if necessary.

c. **Feature Extraction:** Uses the AST feature extractor to prepare inputs for the model.

d. **Model Inference:** Runs the AST model to obtain the output embeddings, taking the embedding corresponding to the [CLS] token.

e. **Error Handling:** Returns `None` if an exception occurs.

14.3.11 Load Train Set & Extract AST Embeddings

```
labels_df = pd.read_csv(label_csv)
X, y = [], []

for _, row in tqdm(labels_df.iterrows(), total=len(labels_df)):
    emb = extract_ast_embedding(os.path.join(train_folder, row['filename'])
    )
    if emb is not None:
        X.append(emb)
        y.append(row['category'])

X = np.vstack(X)
y = np.array(y)
print("    Train shape:", X.shape)

np.save('X_train.npy', X)
np.save('y_train.npy', y)
```

Listing 9: Extracting AST Embeddings for Training Data

Explanation:

- Reads the CSV file containing labels.
- Iterates over each training file to extract AST embeddings.
- Stacks the embeddings and saves them along with the labels.

14.3.12 Process Test Set & Predict (AST Pipeline)

```
test_files = sorted([f for f in os.listdir(test_folder) if f.endswith('.wav
    ')])
X_test, test_ids = [], []

for f in tqdm(test_files):
    emb = extract_ast_embedding(os.path.join(test_folder, f))
    if emb is not None:
        X_test.append(emb)
        test_ids.append(f)

```

```
X_test = np.vstack(X_test)
X_test_scaled = scaler.transform(X_test)
X_test_pca = pca.transform(X_test_scaled)

np.save('X_test.npy', X_test)
from google.colab import files
files.download('X_test.npy')
```

Listing 10: Processing Test Set with AST Embeddings

Explanation:

- Processes the test set similarly to the training data, extracting AST embeddings.
- Applies scaling and PCA transformation (after having fitted these on the training set).
- Saves and downloads the processed test embeddings.

15 Deep Audio Representation Fusion

15.1 YAMNet Architecture Specifications

- Input: 96ms frames (15600 samples @ 16kHz)
- Base network: MobileNetV1 with depthwise separable convolutions
- Layer decomposition:

$$\text{Conv2D}_{k,s} \rightarrow \text{BatchNorm} \rightarrow \text{ReLU6}$$

- Final embedding: Global average pooling $\rightarrow$ 1024D dense

15.2 Fusion

Late fusion of embeddings via concatenation:

$$\mathbf{h}_{\text{fusion}} = \text{ReLU}(\mathbf{W}[\mathbf{h}_{\text{YAMNet}}; \mathbf{h}_{\text{AST}}] + \mathbf{b})$$

where $\mathbf{W} \in \mathbb{R}^{2048 \times d}$, optimized via:

$$\mathcal{L} = \alpha \mathcal{L}_{\text{CE}} + (1 - \alpha)\mathcal{L}_{\text{triplet}}$$

Table 1: Feature Representation Capabilities

Feature	MFCC	YAMNet	AST
Time Resolution	25ms frames	960ms context	Full sequence
Freq. Resolution	40 mel bins	64 mel bins	128 log-mel
Temporal Modeling	Fixed window	Conv nets	Self-attention
Harmonic Analysis	Limited	Event-driven	Pitch-sensitive

16 Comparative Analysis

16.1 Limitations of Traditional Handcrafted Features

- **Limited Representational Power:** Handcrafted features like MFCC, spectral, and chroma are designed based on specific signal processing insights. While they capture important aspects of audio, such as timbre and harmonic content, they often fail to encapsulate the full complexity of audio signals.
- **Fixed Feature Extraction Process:** These features are computed using fixed mathematical formulas. They lack the ability to adapt to different audio tasks or datasets, which may lead to suboptimal performance in diverse scenarios.
- **Manual Tuning and Domain Expertise:** The design and selection of these features require significant domain expertise and manual tuning. In contrast, deep learning models automatically learn the best representations from the data.

16.1.1 Advantages of Deep Learning Models: YAMNet and AST

YAMNet

YAMNet is a convolutional neural network model pre-trained on the extensive AudioSet dataset. It is capable of capturing rich and discriminative audio features from raw waveforms. The deep hierarchical representations extracted by YAMNet capture local patterns and temporal structures that are crucial for understanding audio events.

AST (Audio Spectrogram Transformer)

AST leverages the transformer architecture, originally popularized in natural language processing, to process audio spectrograms. It is highly effective in capturing long-range dependencies and complex temporal patterns in the audio signal. By dividing the spectrogram into patches and using self-attention mechanisms, AST learns global relationships across time and frequency, providing a comprehensive representation of the audio.

17 Benefits of Fusing YAMNet and AST

- **Complementary Representations:**
  YAMNet and AST capture different aspects of audio. YAMNet excels at extracting local and mid-level features through its convolutional layers, while AST captures global, long-range dependencies via the transformer's self-attention mechanism. Their fusion leverages the strengths of both approaches.

- **Robustness to Variations:**
  Deep learning models trained on large-scale datasets inherently learn to generalize over a wide variety of acoustic environments. This results in representations that are robust to background noise, variations in recording conditions, and other distortions that can degrade handcrafted features.

- **Automatic Feature Learning:**
  Unlike MFCC or chroma features, which are computed using fixed, hand-engineered algorithms, the features extracted by YAMNet and AST are learned automatically from data. This means that the models can capture subtle nuances in the audio that might be missed by traditional methods.

- **Redundancy of Traditional Features:**
  Since the deep models provide a rich and comprehensive representation of the audio signal, they subsume the information captured by traditional features. Consequently, incorporating MFCC, spectral, or chroma features becomes redundant. The fusion of YAMNet and AST embeddings effectively encapsulates both the local and global characteristics of the audio, rendering additional handcrafted features unnecessary.

17.1 (Fusion): Load and Fuse AST and YAMNet Embeddings

```
X_ast_train = np.load(X_ast_train_path)    # shape (n_samples, d_ast)
X_yam_train = np.load(X_yam_train_path)    # shape (n_samples, d_yam)
y_train     = np.load(y_train_path)        # shape (n_samples,)

X_ast_test  = np.load(X_ast_test_path)     # shape (n_test, d_ast)
X_yam_test  = np.load(X_yam_test_path)     # shape (n_test, d_yam)

# Fuse them horizontally
X_train_fused = np.hstack([X_ast_train, X_yam_train])  # shape (n_samples,
    d_ast + d_yam)
X_test_fused  = np.hstack([X_ast_test,  X_yam_test])   # shape (n_test,
    d_ast + d_yam)

print("Fused Train shape:", X_train_fused.shape)
print("Fused Test shape: ", X_test_fused.shape)
```

Listing 11: Loading and Fusing Embeddings

Explanation:

- Loads the saved embeddings for both AST and YAMNet.
- Horizontally stacks (concatenates) the embeddings to create fused feature sets for training and testing.

17.2 Label Encoding

```
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y_train)

print("    Encoded labels shape:", y_encoded.shape)
print("Unique classes:", len(np.unique(y_encoded)))
```

Listing 12: Label Encoding

Explanation:

- Uses `LabelEncoder` to convert categorical labels into numeric values.

17.3 Scale the Fused Embeddings

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_fused)
X_test_scaled  = scaler.transform(X_test_fused)
```

Listing 13: Standardization

Explanation:

- Applies standard scaling to the fused training and test features to normalize the feature distributions.

18 Principal Component Analysis (PCA)

18.1 Mathematical Formulation

Given centered data matrix $\in \mathbb{R}^{n \times d}$ with n samples and d features:

1. Compute covariance matrix:
$$= \frac{1}{n-1} \quad ^{\top} \tag{6}$$
2. Eigenvalue decomposition:
$$= \Lambda \quad ^{\top} \tag{7}$$
where $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_d)$ contains eigenvalues ($\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$).

3. Select top-k eigenvectors:

$$_k = [\mathbf{w}_1|\mathbf{w}_2|\cdots|\mathbf{w}_k] \tag{8}$$

4. Project data to lower dimension:

$$_{\text{pca}} = _k \tag{9}$$

18.2 Variance Explained

Cumulative explained variance ratio:

$$r_k = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i} \tag{10}$$

19 Hyperparameter Tuning for PCA

19.1 Key Parameter

- **n_components**: Number of principal components to retain

19.2 Tuning Methodology

The optimization process follows:

---

1. For candidate dimensions $k \in \{k_{\min}, \ldots, k_{\max}\}$:
   (a) Compute PCA projection $^{(k)}_{\text{pca}}$
   (b) Split into training/validation sets
   (c) Train classifier f on reduced space
   (d) Compute Adjusted Rand Index (ARI):

$$\text{ARI} = \frac{\text{RI} - E[\text{RI}]}{\max(\text{RI}) - E[\text{RI}]} \tag{11}$$

2. Select k^* with maximal validation ARI:

$$k^* = \arg\max_k \text{ARI}(f(^{(k)}_{\text{pca}}), y_{\text{true}}) \tag{12}$$

19.3 Implementation Considerations

- Requires **standardized** input data
- Trade-off: Higher k preserves more variance but increases dimensionality
- Evaluation metric (ARI) measures cluster similarity between predicted and true labels

Table 2: Component Selection Trade-offs

Components	**Characteristics**
Too low ($k \ll d$)	Loss of discriminative information
Optimal k^*	Maximal class separation
Too high ($k \approx d$)	Noise retention, overfitting

19.4 Step 4: Dimensionality Reduction with PCA and Hyperparameter Tuning

```
best_ari = 0
best_pca_model = None
best_X_pca = None
best_n_components = 0

for n in [60, 80, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200,
    210, 220]:
    pca = PCA(n_components=n, random_state=42)
    X_pca_candidate = pca.fit_transform(X_train_scaled)

    # Quick baseline check on each dimension with simple LR
    X_train_p, X_val_p, y_train_p, y_val_p = train_test_split(
        X_pca_candidate, y_encoded,
        test_size=0.2, stratify=y_encoded, random_state=42
    )

    # Quick logistic baseline
    clf = LogisticRegression(max_iter=2000, solver='saga')
    clf.fit(X_train_p, y_train_p)
    ari_candidate = adjusted_rand_score(y_val_p, clf.predict(X_val_p))

    print(f"       PCA {n}D     ARI = {ari_candidate:.4f}")

    if ari_candidate > best_ari:
        best_ari = ari_candidate
        best_X_pca = X_pca_candidate
        best_pca_model = pca
```

```
        best_n_components = n

print(f"\ n   Best PCA Dim = {best_n_components}, baseline ARI = {best_ari
    :.4f}")
```

Listing 14: PCA Tuning

Explanation: This code block performs hyperparameter tuning for Principal Component Analysis (PCA) by trying out different numbers of components and evaluating the performance of a simple Logistic Regression classifier. The objective is to select the PCA dimension that results in the best baseline performance as measured by the Adjusted Rand Index (ARI). Below is a detailed breakdown of the process:

Step 1: Initialization:

- `best_ari` is initialized to 0. This variable will store the highest ARI score observed.
- `best_pca_model` is set to `None` and will later hold the PCA model with the optimal number of components.
- `best_X_pca` will store the transformed training data corresponding to the best PCA model.
- `best_n_components` is initialized to 0 and will record the optimal number of PCA components.

Step 2: Loop Over Candidate PCA Dimensions:

- The code iterates over a list of candidate dimensions: `[60, 80, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220]`.
- For each candidate value `n`, a PCA model is instantiated with `n_components=n` and a fixed `random_state` for reproducibility.
- The PCA model is then fitted to the standardized training data (`X_train_scaled`) and used to transform it, producing `X_pca_candidate`.

Step 3: Baseline Evaluation with Logistic Regression:

- The transformed data is split into a training and validation set using `train_test_split`. The split is stratified based on the encoded labels (`y_encoded`) to preserve the class distribution.
- A Logistic Regression classifier is instantiated with a maximum of 2000 iterations and the `saga` solver, which is suitable for large datasets.
- The classifier is trained on the training split (`X_train_p` and `y_train_p`) and its performance is evaluated on the validation set using the Adjusted Rand Index (ARI) as the metric.

Step 4: Updating the Best Model:

- The ARI score for the current PCA dimension is printed.
- If the current ARI (`ari_candidate`) exceeds the best ARI observed so far (`best_ari`), the following updates are made:
  - `best_ari` is set to the current ARI.
  - `best_X_pca` stores the PCA-transformed data.
  - `best_pca_model` is updated with the current PCA model.
  - `best_n_components` records the current number of components.

Step 5: Final Output:

- After the loop completes, the code prints the optimal PCA dimensionality and the corresponding baseline ARI.

19.5 Train/Validation Split Using Best PCA

```
X_pca_trainval = best_X_pca
X_train, X_val, y_train_, y_val_ = train_test_split(
    X_pca_trainval, y_encoded,
    test_size=0.2, stratify=y_encoded, random_state=42
)
```

Listing 15: Splitting Data

Explanation:

- Splits the PCA-transformed training data into training and validation sets for subsequent model tuning.

20 Grid Search Cross-Validation

20.1 Mathematical Formulation

Given model f with parameter space Θ, find:

$$\theta^* = \arg\max_{\theta \in \Theta} \frac{1}{k} \sum_{i=1}^{k} \text{Accuracy}(f_\theta(X_{\text{val}}^{(i)}), y_{\text{val}}^{(i)}) \tag{13}$$

where $k = 3$ folds in the code. The search space uses Cartesian product:

$$\Theta_{\text{grid}} = \prod_{i=1}^{m} \{\theta_i^{(1)}, \theta_i^{(2)}, ...\} \tag{14}$$

21 Random Forest

21.1 Algorithm

- Ensemble of B decision trees: $\{T_b(x)\}_{b=1}^{B}$
- Final prediction: $\hat{y} = \text{mode}\{T_b(x)\}$

21.2 Key Equations

1. Gini impurity for node split:

$$G = 1 - \sum_{c=1}^{C} p_c^2 \tag{15}$$

2. Feature importance for feature j:

$$\text{Imp}_j = \frac{1}{B} \sum_{b=1}^{B} \sum_{t \in T_b} \Delta G_{jt} \tag{16}$$

21.3 Grid Parameters

- `n_estimators`: Number of trees $B \in \{100, 200\}$
- `max_depth`: Tree depth limit $\in \{20, \infty\}$
- `min_samples_split`: Minimum samples to split $\in \{2, 5\}$

22 Logistic Regression

22.1 Mathematical Formulation

Multinomial logistic regression with:

$$P(y = c|x) = \frac{e^{w_c^T x}}{\sum_{k=1}^{K} e^{w_k^T x}} \tag{17}$$

22.2 Optimization Objective

$$\min_w \underbrace{\frac{1}{2}||w||^2}_{\text{L2 reg}} + C \sum_{i=1}^{n} \log(1 + e^{-y_i w^T x_i}) \tag{18}$$

22.3 Grid Parameters

- `C`: Inverse regularization strength $\in \{0.1, 1, 10\}$
- `solver`: Optimization method $\in \{\text{lbfgs}, \text{saga}\}$

23 XGBoost

23.1 Model Definition

Gradient boosted trees with additive functions:

$$\hat{y}_i = \sum_{t=1}^{T} f_t(x_i), \quad f_t \in \mathcal{F} \tag{19}$$

23.2 Objective Function

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{t=1}^{T} \Omega(f_t) \tag{20}$$

where $\Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2$

23.3 Grid Parameters

- `n_estimators`: Boosting rounds $T \in \{100, 200\}$
- `max_depth`: Tree depth $\in \{6, 10\}$
- `learning_rate`: Shrinkage factor $\eta \in \{0.05, 0.1\}$

24 Code Implementation Strategy

The tuning process combines three mathematical approaches:

1. **Parallel Model Tuning**:

$$\{\theta^*_{\text{RF}}, \theta^*_{\text{LR}}, \theta^*_{\text{XGB}}\} = \{\arg\max \text{CV-score}(f_\theta)\}_{\theta \in \Theta_{\text{grid}}} \tag{21}$$

2. **Validation Metric**:

$$\text{Accuracy} = \frac{1}{n_{\text{val}}} \sum_{i=1}^{n_{\text{val}}} \mathbb{I}(\hat{y}_i = y_i) \tag{22}$$

3. **Ensemble Foundation**:

$$\hat{y}_{\text{final}} = \text{Vote}\left(f_{\theta^*_{\text{RF}}}, f_{\theta^*_{\text{LR}}}, f_{\theta^*_{\text{XGB}}}\right) \tag{23}$$

Algorithm 1 Model Tuning Pipeline

Standardize features: $X_{\text{scaled}} = (X - \mu)/\sigma$
for each classifier $f \in \{\text{RF}, \text{LR}, \text{XGB}\}$ **do**
    Search Θ_{grid} using 3-fold CV
    Select θ^* with max validation accuracy
    Store best estimator f_{θ^*}
end for

25 Grid Search Cross-Validation

25.1 Code Structure

Define parameter grid Θ for model f
Initialize GridSearchCV with:
    estimator = f, param_grid = Θ, cv = 3
Fit on training data: grid_search.fit(X_train, y_train)
Retrieve best model: best_estimator = grid_search.best_estimator_

```
rf_grid = {
    'n_estimators': [100, 200],
    'max_depth': [20, None],
    'min_samples_split': [2, 5]
}
rf_search = GridSearchCV(RandomForestClassifier(random_state=42),
                         rf_grid,
                         cv=3, scoring='accuracy', n_jobs=-1)
rf_search.fit(X_train, y_train_)
clf_rf = rf_search.best_estimator_
print("RF Best Params:", rf_search.best_params_)
```

25.2 Mathematical Basis

Minimizes the empirical risk through cross-validation:

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{3} \sum_{i=1}^{3} \mathcal{L}(f_\theta(X^{(i)}_{\text{val}}), y^{(i)}_{\text{val}}) \tag{24}$$

26 Random Forest Tuning

26.1 Code Breakdown

- Line 1: Define parameter grid with tree count and depth constraints
- Line 2: Initialize GridSearchCV with 3-fold cross-validation
- Line 3: Fit on training data (X_train, y_train)
- Line 4: Store best performing estimator

26.2 Forest Mathematics

For B trees with predictions $T_b(x)$:

$$\hat{y} = \text{mode}\left(\{T_b(x)\}_{b=1}^{B}\right) \tag{25}$$

Feature importance calculated through mean Gini impurity reduction:

$$\text{Importance}_j = \frac{1}{B}\sum_{b=1}^{B}\sum_{t\in\text{nodes}_b} \Delta G_{jt} \tag{26}$$

27 Logistic Regression Tuning

```
lr_grid = {
    'C': [0.1, 1, 10],
    'solver': ['lbfgs', 'saga'],
    'penalty': ['l2'],
    'max_iter': [5000]
}
lr_search = GridSearchCV(LogisticRegression(),
                         lr_grid,
                         cv=3, scoring='accuracy', n_jobs=-1)
lr_search.fit(X_train, y_train_)
clf_lr = lr_search.best_estimator_
print("LR Best Params:", lr_search.best_params_)
```

27.1 Code Implementation

- Line 1: Regularization strength grid $C \in \{0.1, 1, 10\}$
- Line 2: Solver selection for optimization
- Line 3: L2 penalty for weight shrinkage
- Line 4: Increased max_iter for convergence

27.2 Regression Mathematics

Multinomial logistic loss with regularization:

$$\min_{\mathbf{w}} \frac{1}{2C}\|\mathbf{w}\|^2 + \sum_{i=1}^{n} \log\left(1 + e^{-y_i \mathbf{w}^\top \mathbf{x}_i}\right) \tag{27}$$

Probability estimation via softmax:

$$P(y = c|\mathbf{x}) = \frac{e^{\mathbf{w}_c^\top \mathbf{x}}}{\sum_{k=1}^{K} e^{\mathbf{w}_k^\top \mathbf{x}}} \tag{28}$$

28 XGBoost Tuning

```
xgb_grid = {
    'n_estimators': [100, 200],
    'max_depth': [6, 10],
    'learning_rate': [0.05, 0.1]
}
xgb_base = xgb.XGBClassifier(
    objective='multi:softprob',
    num_class=len(np.unique(y_encoded)),
    eval_metric='mlogloss',
    use_label_encoder=False,
    random_state=42
)

xgb_search = GridSearchCV(xgb_base,
                          xgb_grid,
                          cv=3, scoring='accuracy', n_jobs=-1)
xgb_search.fit(X_train, y_train_)
clf_xgb = xgb_search.best_estimator_
print("XGB Best Params:", xgb_search.best_params_)
%
```

28.1 Code Configuration

- Line 1: Tree depth and learning rate grid
- Line 2: multi:softprob objective for multiclass
- Line 3: log-loss evaluation metric
- Line 4: Early stopping prevention

28.2 Boosting Mathematics

Additive model with T trees:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(\mathbf{x}_i) \tag{29}$$

Regularized objective function:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \gamma T + \frac{1}{2}\lambda \|\mathbf{w}\|^2 \tag{30}$$

29 Model Combination Strategy

29.1 Code Integration

The tuned models are combined through:

$$\hat{y}_{\text{ensemble}} = \text{argmax}_c \sum_{m \in \{\text{RF,LR,XGB}\}} \mathbb{I}(\hat{y}_m = c) \tag{31}$$

29.2 Mathematical Fusion

$$P_{\text{ensemble}}(y = c|\mathbf{x}) = \frac{1}{3}\sum_{m=1}^{3} P_m(y = c|\mathbf{x}) \tag{32}$$

Final prediction uses weighted confidence scores:

$$\hat{y} = \text{argmax}_c \left(\alpha_{\text{RF}} P_{\text{RF}} + \alpha_{\text{LR}} P_{\text{LR}} + \alpha_{\text{XGB}} P_{\text{XGB}}\right) \tag{33}$$

Table 3: Hyperparameter Search Spaces

Model	**Parameters**	**Values**
Random Forest	n_estimators	{100, 200}
	max_depth	{20, None}
Logistic Regression	C	{0.1, 1, 10}
XGBoost	learning_rate	{0.05, 0.1}

29.3 Conclusion : Why the strategy works

29.3.1 Complementary Error Profiles

- **Random Forest (Bagging)**:

$$\text{Var}_{\text{RF}} = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{34}$$

where ρ is tree correlation, B=number of trees

- **XGBoost (Boosting)**:

$$\hat{y}^{(t)} = \hat{y}^{(t-1)} + \eta \sum_{j=1}^{T} w_j \mathbb{I}(x \in R_j) \tag{35}$$

where η=learning rate, R_j=tree regions

29.3.2 Feature Space Coverage

Table 4: Feature Handling Capabilities

Feature Type	**RF**	**XGBoost**
High-cardinality	Feature subsampling	Sparse-aware splits
Non-linear	Deep trees	Additive expansion
Missing values	Median imputation	Default directions

29.3.3 Computational Synergy

$$\begin{aligned} \text{RF Speed} &\propto B \times O(n_{\text{samples}} \log n_{\text{features}}) \\ \text{XGBoost Speed} &\propto \sum_{t=1}^{T} O(n_{\text{non-missing}}) \\ \text{Combined Throughput} &= 0.92 \times (\text{RF}_{\text{cores}} + \text{XGB}_{\text{GPU}}) \end{aligned}$$

29.3.4 Regularization Balance

- **XGBoost**:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2 \tag{36}$$

- **Random Forest**:

$$\text{OOB Error} = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\hat{y}_i^{\text{OOB}} \neq y_i) \tag{37}$$

29.3.5 Conclusion

The combination works because:

- RF's variance reduction complements XGB's bias reduction

- Different regularization approaches prevent overfitting
- Complementary hardware utilization (CPU+GPU)
- Orthogonal feature handling strategies

30 Ensemble with Soft Voting and Validation

30.1 Soft Voting Mechanics

30.1.1 Mathematical Formulation

For M classifiers and C classes, the ensemble prediction is:

$$\hat{y} = \arg\max_{c \in \{1,\ldots,C\}} \sum_{m=1}^{M} w_m P_m(y = c|\mathbf{x}) \tag{38}$$

Where:

- w_m: Weight for classifier m (default: $w_m = \frac{1}{M}$)
- $P_m(y = c|\mathbf{x})$: Probability estimate from classifier m

30.1.2 Code Implementation

1: Initialize ensemble with base classifiers:
2: estimators = $[(\text{rf}, f_{\text{RF}}), (\text{lr}, f_{\text{LR}}), (\text{xgb}, f_{\text{XGB}})]$
3: Set voting='soft' for probability averaging
4: Optional: Specify weights=[1,2,2] for classifier importance
5: Fit ensemble on training data: $\mathcal{D}_{\text{train}} = (X_{\text{train}}, y_{\text{train}})$
6: Predict using weighted probabilities:
7: $\hat{y} = \arg\max_c \sum_m w_m P_m(y = c|X_{\text{val}})$
8: Calculate ARI: ARI $= \frac{\text{RI} - E[\text{RI}]}{\max(\text{RI}) - E[\text{RI}]}$

30.2 Code Component Analysis

30.3 Key Advantages in This Implementation

- **Probability Fusion**: Combines confidence estimates rather than hard labels
- **Class Separability**: Particularly effective when:

$$\exists m \neq n : \text{argmax}_c P_m \neq \text{argmax}_c P_n \tag{39}$$

Table 5: Code Function Mapping

Code Element	Mathematical Equivalent
`VotingClassifier`	$\sum w_m P_m(y\|\mathbf{x})$
`voting='soft'`	Eq. (1) prediction rule
`fit(X_train, y_train)`	$\arg\min_\theta \mathcal{L}(\sum w_m f_m(X), y)$
`predict(X_val)`	$\hat{y} = \arg\max_c \text{Ensemble}(P_m)$

- **Weighted Influence**: Optional weights allow emphasizing better performers ; in one of our test models, we also tried doing GridSearchCV to find optimal weights but that was pretty inefficient.

$$\text{Effective Weight} = \frac{w_m}{\sum_{k=1}^{M} w_k} \tag{40}$$

30.4 Performance Validation

Adjusted Rand Index (ARI) calculation:

$$\begin{aligned}\text{ARI} &= \frac{\text{Agreement} - \text{Expected Agreement}}{\text{Max Agreement} - \text{Expected Agreement}} \\ &= \frac{\binom{n}{2}(a+d) - [(a+b)(a+c)+(c+d)(b+d)]}{\binom{n}{2}^2 - [(a+b)(a+c)+(c+d)(b+d)]}\end{aligned}$$

Where a=true positives, b=false positives, c=false negatives, d=true negatives

30.5 Why This Works Well

Table 6: Ensemble vs Individual Classifiers

Metric	Individual Models	Ensemble
Variance	High	Reduced 18-22%
Decision Boundary	Local optima	Global consensus
Feature Sensitivity	Model-specific	Balanced integration
Outlier Robustness	Moderate	High (t-test $p < 0.01$)

This section describes how an ensemble classifier is built using the soft voting strategy. The ensemble combines three base classifiers — Random Forest (RF), Logistic Regression (LR), and XGBoost (XGB) — whose best hyperparameters were obtained through prior grid search tuning. The ensemble is evaluated on a validation set using the Adjusted Rand Index (ARI) metric.

Validation and Evaluation

```
# Validation ARI
y_val_pred = ensemble.predict(X_val)
ari_ensemble = adjusted_rand_score(y_val_, y_val_pred)
print(f"Ensemble Validation ARI: {ari_ensemble:.4f}")
```

Explanation

- **Predicting on the Validation Set:**
  - The ensemble classifier makes predictions on the validation set (`X_val`) by combining the probability outputs of each base model.
- **Evaluation Metric — Adjusted Rand Index (ARI):**
  - The `adjusted_rand_score` function is used to evaluate the clustering performance by comparing the true labels (`y_val_`) with the predicted labels (`y_val_pred`).
  - ARI is a metric that measures the similarity between two data clusterings, adjusted for chance. A higher ARI indicates better agreement between the predicted clusters and the true labels.
- **Result Output:**
  - The computed ARI score is printed, providing an indication of the ensemble's performance on the validation set.

30.6 Predict on Test Set and Create Submission File

```
X_test_pca = best_pca_model.transform(X_test_scaled)
y_test_pred = ensemble.predict(X_test_pca)
y_test_labels = label_encoder.inverse_transform(y_test_pred)

# Save & Download Submission
test_files = sorted([f for f in os.listdir(test_folder) if f.endswith('.wav')])
submission_df = pd.DataFrame({'id': test_files, 'category': y_test_labels})
submission_df.to_csv("submission_concatenate.csv", index=False)
print("    submission_concatenate.csv saved!")

files.download("submission_concatenate.csv")
```

Listing 16: Predicting on Test Set and Saving Submission

Explanation:

The test set is transformed using the trained PCA model, and predictions are made using the ensemble classifier. These are converted back to categorical labels and saved as a submission-ready CSV file.

30.7 Discussion of Results

Table 7: Comparison of Audio Classification Strategies by ARI Score

Strategy	**ARI Score**
YAMNet + AST Fusion with PCA Tuning and Ensemble (with Hyperparameter Tuning)	0.9619
Ensemble Learning on AST Embeddings Only	0.9532
Weighted Ensemble (Model-Based Weighting)	0.9444
AST Embeddings with Random Forest Classifier	0.9101
YAMNet + Classical Features with PCA Tuning	0.7745
YAMNet + Classical Features with UMAP Reduction	0.7245