Speech Recognition

Assignment 1: Speech signal processing

Name:	赵卓冰
Student Number:	2252750
Major:	软件工程
Grade:	2023
Teacher:	沈莹

2024.10.23

1. Overview

Extracting **Mel Frequency Cepstral Coefficients (MFCCs)**
from audio signals is one of the most common feature extraction techniques in audio and speech processing. The implementation captures a comprehensive pipeline starting from loading an audio signal to generating the final feature vectors used in machine learning models for audio classification, speech recognition, and other audio-related tasks.

2. Objectives

- **Pre-process** audio signals with pre-emphasis and framing.
- Apply **Short-Time Fourier Transform (STFT)** for frequency domain analysis.
- Compute the **Mel-filter bank** to simulate human auditory perception.
- Calculate the **MFCC features**, including dynamic features (delta and delta-delta).

- **Visualize** key steps in the pipeline such as signal waveforms, STFT spectrograms, Mel filter banks, and MFCC features.

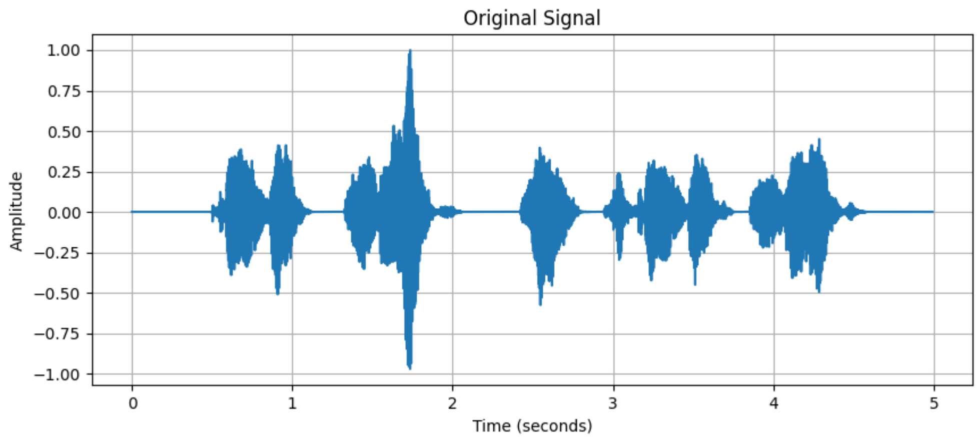
3. Implementation Details

3.1. Recording Parameters

Primarily, for this project, I created a recording called input.wav using my own voice. `input.wav` with my own voice.

- **Format:** `pyaudio.paInt16`
    - The audio is recorded using 16-bit depth, which provides high-quality sound by capturing 65,536 distinct amplitude levels. This setting ensures that the audio is suitable for subsequent analysis, especially when extracting Mel Frequency Cepstral Coefficients (MFCCs) for speech recognition or other tasks.
- **Channels: Mono (`1`)**
    - The recording is performed in mono, which is sufficient for most speech processing tasks. Mono reduces the data size compared to stereo while preserving the essential features of the sound, especially for human speech.
- **Sampling Rate:** `44100` **Hz**
    - The sampling rate is set to 44.1 kHz, which is standard for CD-quality audio. This rate captures the audio signal in enough detail to ensure accurate MFCC extraction and other analyses while keeping the file size manageable.
- **Chunk Size:** `1024` **samples**
    - Audio is processed in chunks of 1024 samples at a time. This chunk size provides a balance between performance and real-time processing. It ensures that the recording stream can handle the audio data efficiently without introducing excessive latency.
- **Recording Duration:** `5 seconds`
    - The main goal of this project is to extract MFCC features and the recording duration is not very important, so 5 seconds is enough.
- **Output Format:** `.wav`

    - The recorded audio is saved as a `.wav` file, an uncompressed format that retains the full fidelity of the recorded sound. This is important for feature extraction as no information is lost during compression.
- **Signal Wave of Orignal Audio**



3.2. Pre-emphasis

The pre-emphasis operation enhances the high-frequency components of an audio signal. This helps balance the audio spectrum and improves feature extraction.

3.2.1. Pre-emphasis Formula:

The pre-emphasis formula is:

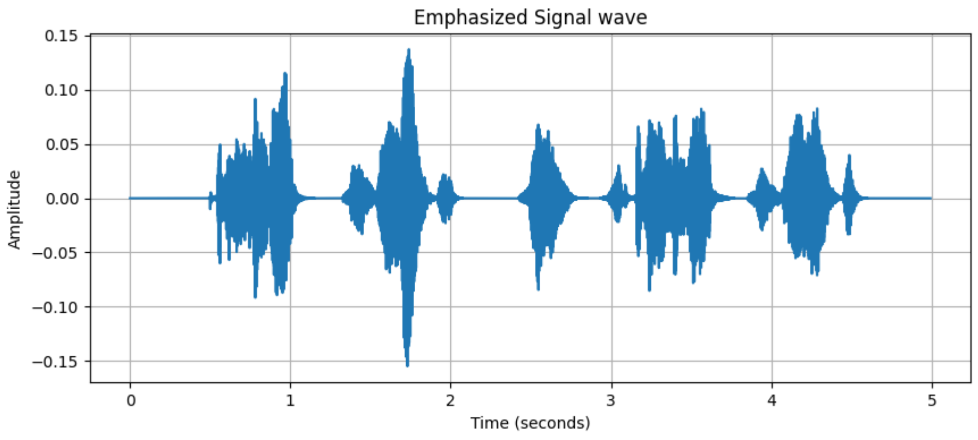
$$y(t) = x(t) - \alpha \cdot x(t-1)$$

where:

- `x(t)` is the audio signal sample at time `t`.
- `x(t-1)` is the sample at the previous time step.
- `α` is the pre-emphasis coefficient, usually around **0.97**, which controls the influence of the previous sample on the current sample.

```python
def pre_emphasis(signal, alpha=0.97):
    emphasized_signal = np.append(signal[0], signal[1:] - alpha * signal[:-1])
    return emphasized_signal
```

- **Emphasized Signal Wave**



3.3. Framing and Windowing

The signal is divided into short frames to capture time-localized information. A Hamming window is applied to each frame to minimize signal discontinuities at the boundaries.

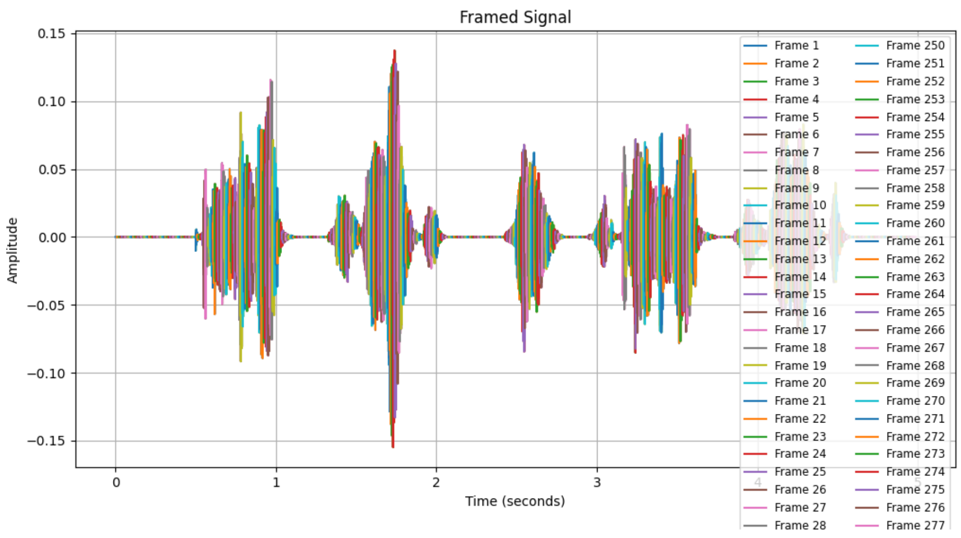
```python
def windowing(frames, frame_length):
    hamming_window = np.hamming(frame_length)
    windowed_frames = frames * hamming_window
    return windowed_frames
```

3.4. Short-Time Fourier Transform (STFT)

STFT is applied to convert the signal into the frequency domain. The magnitude of the Fourier transform is taken for each frame:

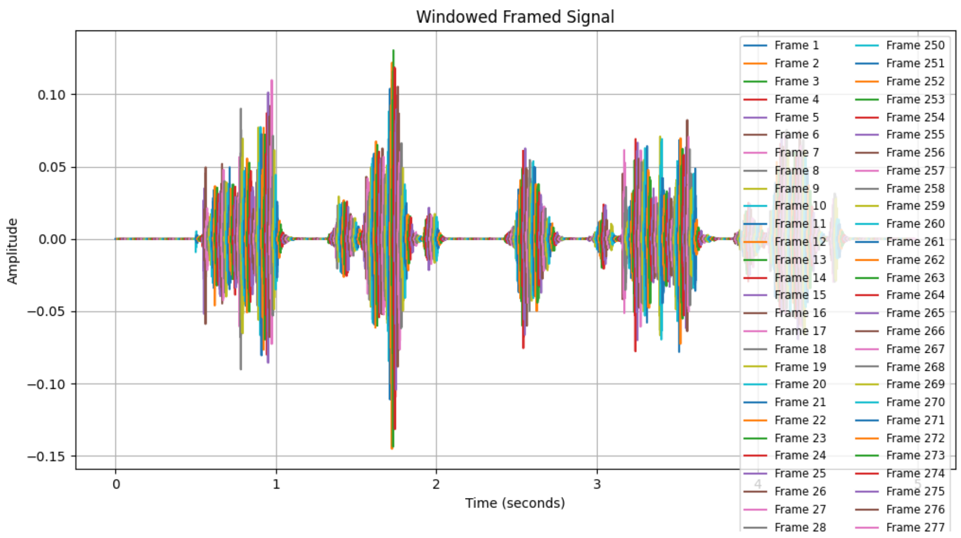
```python
    # Step 2: Framing
    signal_length = len(emphasized_signal)
    # Ceil to promise the possible remaining part can be a frame (padding by zero)
    num_frames = int(np.ceil(float(np.abs(signal_length - frame_length)) / frame_step))
    # Signal length after padding by zero
    pad_signal_length = num_frames * frame_step + frame_length
    z = np.zeros(pad_signal_length - signal_length)
    # Signal after padding by zero
    pad_signal = np.append(emphasized_signal, z)
    # Generate frame indices matrix
    indices = np.tile(np.arange(0, frame_length), (num_frames, 1)) + \
              np.tile(np.arange(0, num_frames * frame_step, frame_step), (frame_length, 1)).T
    # Extract sample points from signal. frames is a matrix, 
in which every row is a frame.
    frames = pad_signal[indices.astype(np.int32, copy=False)]
    plot_frames(frames, frame_length, frame_step, sample_rate, 
"Framed Signal")
```

- **Framed Signal**



```
def compute_stft(windowed_frames, NFFT=2048):
    magnitude_frames = np.abs(np.fft.rfft(windowed_frames, 
NFFT))
    return magnitude_frames
```

- **Windowed Signal**



3.5. Mel Filter Bank

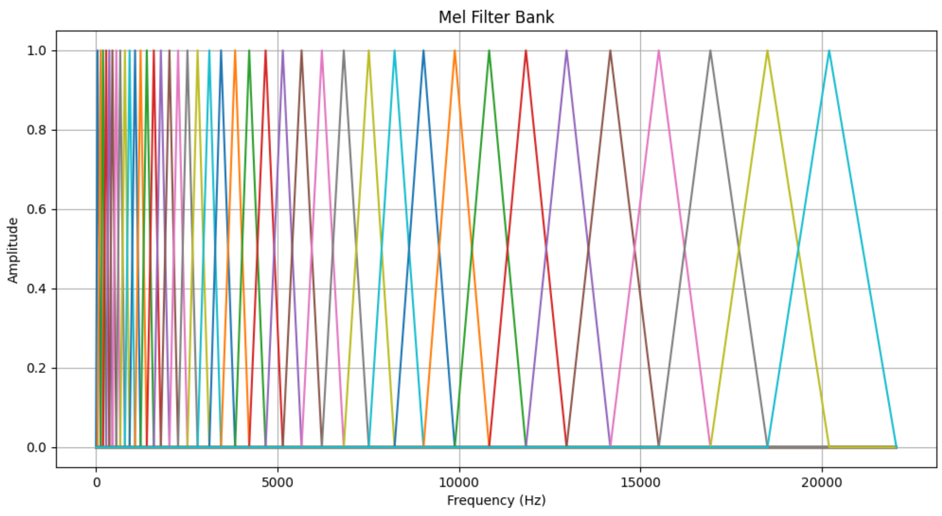
The Mel scale filter bank is created to simulate human hearing sensitivity, which focuses more on lower frequencies:

```python
# Function to calculate Mel-filter bank
def mel_filter_bank(num_filters, NFFT, sample_rate):
    # Mel scale reflects the perception of frequency by human ear which has a finer perception of the low frequencies.
    # Mel scale of lowest frequency (i.e. 0 Hz)
    low_freq_mel = 0
    # Mel scale of highest frequency (i.e. nyquist frequency, half of sample rate for any signal)
    high_freq_mel = 2595 * np.log10(1 + (sample_rate / 2) / 700)
    # Generates points on the mel frequency scale
    # Plus 2 because the lowest_freq_mel point and highest_freq_mel point need to be included.
    mel_points = np.linspace(low_freq_mel, high_freq_mel, num_filters + 2)
    # Convert Mel scale back to Hz
    hz_points = 700 * (10**(mel_points / 2595) - 1)
    # Convert Hz to Fourier transform spectrum bin
    bin_points = np.floor((NFFT + 1) * hz_points / sample_rate).astype(int)

    # Create a filter bank matrix
    filter_bank = np.zeros((num_filters, int(NFFT / 2 + 1)))

    # Generate triangle filter
    for i in range(1, num_filters + 1):
        filter_bank[i - 1, bin_points[i - 1]:bin_points[i]] = \
            (np.arange(bin_points[i - 1], bin_points[i]) - bin_points[i - 1]) / (bin_points[i] - bin_points[i - 1])
        filter_bank[i - 1, bin_points[i]:bin_points[i + 1]] = \
            (bin_points[i + 1] - np.arange(bin_points[i], bin_points[i + 1])) / (bin_points[i + 1] - bin_points[i])
    return filter_bank
```

- **Mel Filter Bank result**

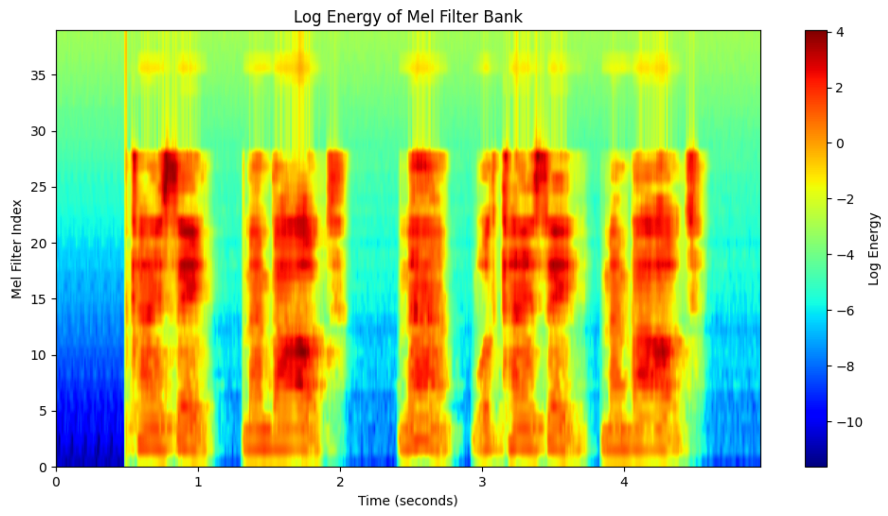


3.6. Log Energy of Filter Bank

The energy output of the Mel filter bank is logarithmically scaled to enhance signal dynamics:

```
# Step 6: Taking logarithm of filter bank energies
# Replace 0 with a small value
filter_bank_energy = np.where(filter_bank_energy == 0,
np.finfo(float).eps, filter_bank_energy)
log_energy =np.log(filter_bank_energy)
plot_log_energy(log_energy, sample_rate, frame_step, title="Log
Energy of Mel Filter Bank")
```

- **Log Energy of Filter Bank Result**



3.7. Discrete Cosine Transform (DCT)

DCT is applied to the log energies to derive the MFCC features:

```python
# Step 7: Apply Discrete Cosine Transform (DCT) to getM MFCC (12 coefficients)
mfcc_coefficients = dct(log_energy, type=2, axis=1, norm='ortho')[:, :num_ceps]
```

This extracts 12 MFCC coefficients, which represent the most significant aspects of the sound's spectral envelope.

3.8. Adding Energy and Dynamic Features

The energy of each frame is added to the MFCC as the 13th feature, and first and second-order derivatives (delta and delta-delta) are calculated to capture dynamic features:

```python
# Step 8: Calculate energy (add energy to MFCCs)
energy = np.sum(windowed_frames ** 2, axis=1)  # Energy for each frame
mfcc_with_energy = np.hstack([mfcc_coefficients, energy[:, np.newaxis]])  # Add energy as 13th feature
```

3.9. Dynamic Feature Extraction (Delta and Delta-Delta)

In addition to the static MFCC features, dynamic features, such as **Delta** and **Delta-Delta** coefficients, are extracted to capture the temporal variations in the signal over time. These features provide valuable information about the changes in the spectral characteristics, which helps in recognizing patterns that are not apparent from the static MFCCs alone.

```python
# Step 9: Dynamic feature extraction (Delta and Delta-Delta)
delta_mfcc = np.gradient(mfcc_with_energy, axis=0)
delta_delta_mfcc = np.gradient(delta_mfcc, axis=0)
```

By including both delta and delta-delta features, we capture not only the static spectral information but also the temporal dynamics, which are essential for tasks such as speech recognition where the rate and acceleration of sound changes convey important information.

3.10. Feature Transformation

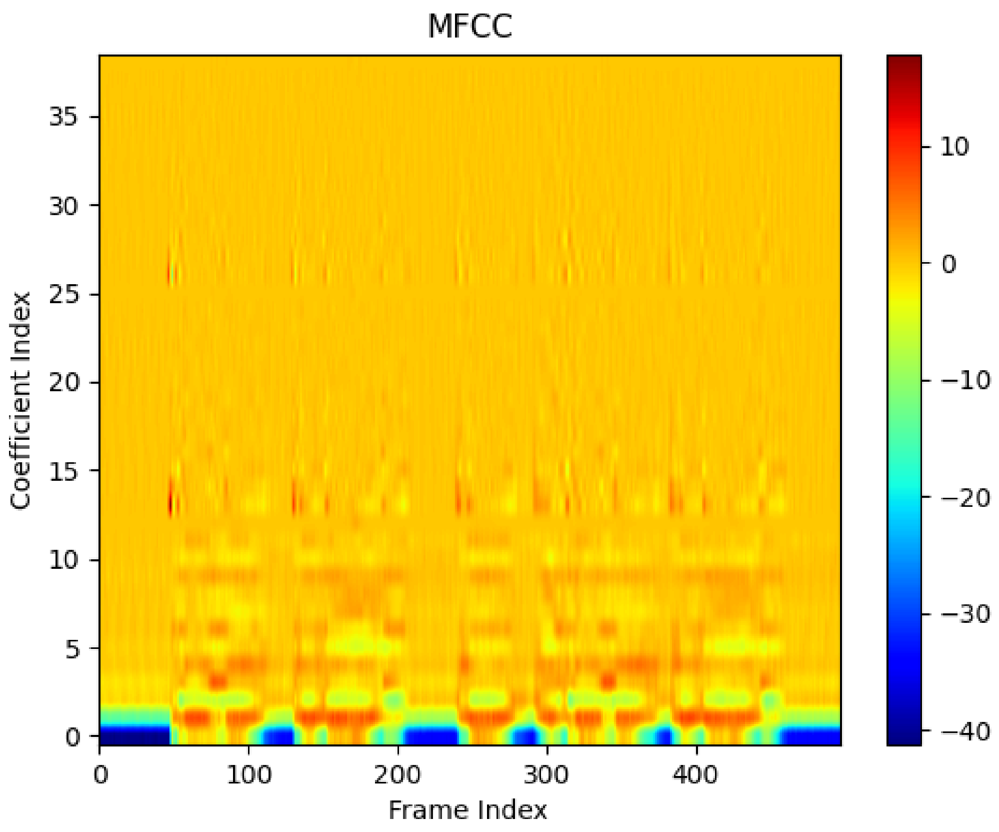
Once the **MFCC** coefficients, **Delta**, and **Delta-Delta** features are computed, they are concatenated to form a comprehensive feature vector. This final step produces a feature set that encapsulates both the static and dynamic properties of the audio signal.

- The static MFCC features represent the spectral characteristics of each frame.
- Delta features provide the velocity or rate of change of these spectral characteristics.
- Delta-Delta features give the acceleration or rate of change of the Delta coefficients.

```
# Step 10: Feature transformation (concatenate MFCC, delta, and delta-delta)
combined_feature = np.hstack([mfcc_with_energy, delta_mfcc, delta_delta_mfcc])
```

This combined feature set provides a rich representation of the audio signal, ready for input into machine learning models for tasks such as speech recognition, speaker identification, or emotion detection.

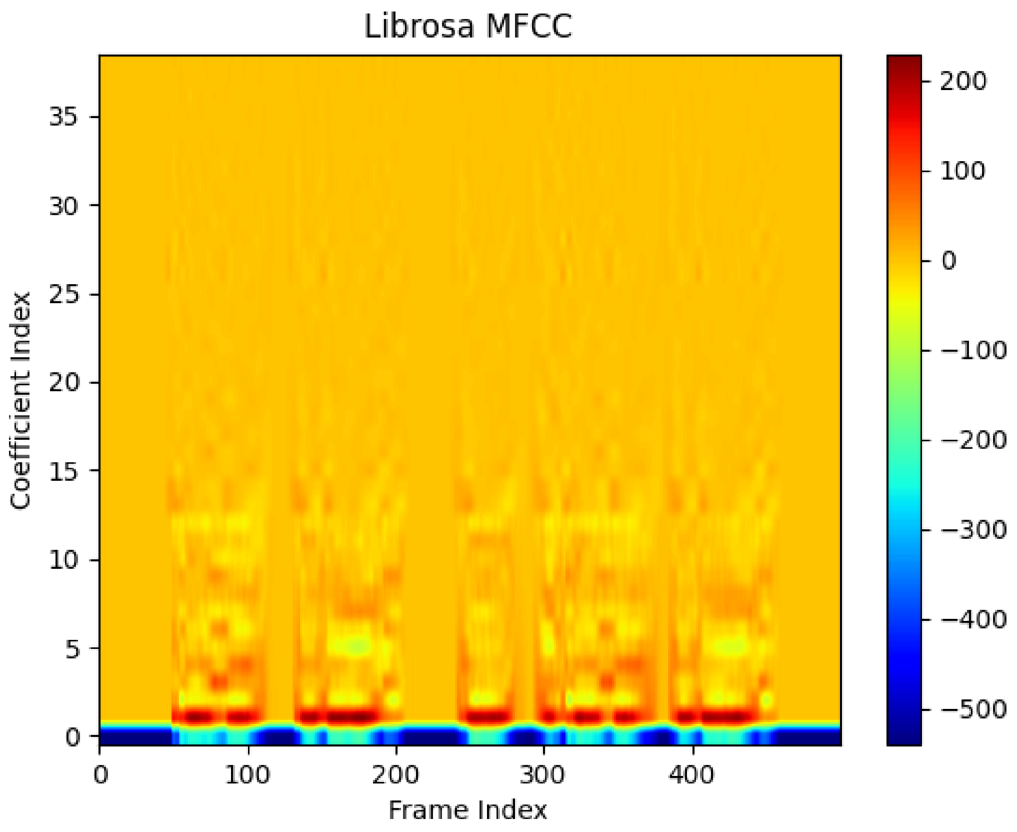
3.11. Final MFCC Features Result



4. Compare With Python Package

Here we use `librosa.feature.mfcc` function to get features directly.

4.1. Librosa MFCC Features Result



4.2. Analysis

The overall trend are the same, and most of the data are similar.

The magnitude differences between my custom MFCC extraction and the results from librosa primarily stem from differences in implementation details and data processing methods. By learn `librosa`'s `mfcc()` function, I got some possible reasons for the magnitude differences:

- **Pre-emphasis**: Since `librosa` does not perform this operation, the frequency components are used directly for feature extraction, leading to different feature values and magnitudes.
- **Scaling and Normalization**: `librosa`'s `mfcc()` function might apply certain normalization steps to the computed MFCC feature values, resulting in differences in scale compared to my manual implementation. Normalization helps to control the range of features and prevents model training issues related to large feature magnitudes.
- **FFT Size (NFFT)**: The number of FFT points (NFFT) in librosa may differ from what I have set in my implementation. The FFT size directly affects the frequency resolution, which, in turn, impacts the calculation of the filter bank energies and the resulting MFCC values.

5. Future Improvements

- **Noise reduction**: Incorporating denoising techniques to improve feature quality for noisy signals.
- **Real-time Processing**: Enhance the system to process audio in real-time for live applications.

6. Summary

Through this project, I gained a deeper understanding of how **MFCC (Mel Frequency Cepstral Coefficients)** are extracted from audio signals and their importance in tasks such as speech recognition and audio classification. By manually implementing each step of the MFCC extraction process—ranging from pre-processing and framing, to applying the Short-Time Fourier Transform (STFT), generating the Mel filter bank, and calculating dynamic features like delta and delta-delta—I developed a strong grasp of the intricacies involved in feature extraction from audio data.