Assignment 1: MFCC - Report

written by 2051973 韩嘉睿 软件工程 on Oct 18, 2023

In this assignment, we delve into the process of extracting Mel Frequency Cepstral Coefficients (MFCC) from audio signals. MFCCs are widely used in the field of automatic speech recognition (ASR) and are essential features for speech processing tasks. This report will provide a brief overview of the MFCC extraction process, highlight key steps, and showcase critical output figures.

Audio File Reading:

To commence the MFCC extraction process, a Chinese audio segment of approximately 13 seconds in duration was recorded. This audio sample, recorded specifically for this assignment, was saved as 'mfcc.wav'. We began by reading the audio file 'mfcc.wav', identifying its sampling rate as 48000.

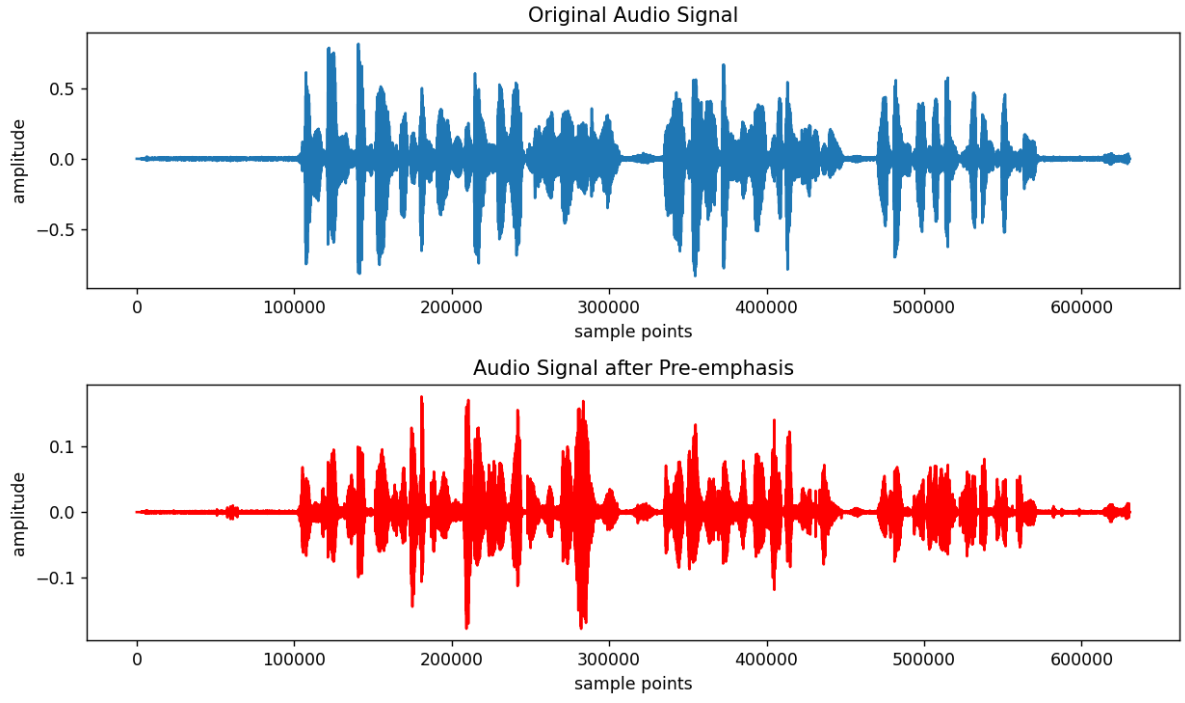
```
# read voice file 'mfcc.wav'
y, sr = librosa.load('mfcc.wav', sr=None)   # the sampling rate of this file is
48000
```

Pre-emphasis

To enhance high-frequency components and reduce decay in the signal, we applied pre-emphasis to the raw audio signal. This step involved applying a first-order pre-emphasis filter with a pre-defined coefficient of 0.97 to the audio signal.

```
def pre_emphasis(signal, coefficient=0.97):
    return np.append(signal[0], signal[1:] - coefficient * signal[:-1])
y_preemphasized = pre_emphasis(y)
```

A pair of subplots visualizes is used to demonstrate the effect of pre-emphasis on the original audio signal.



Windowing

In this step, we focused on framing and windowing the pre-emphasized audio signal.

We set the frame size to 0.025 seconds and the frame stride to 0.01 seconds. We then computed the frame length and frame step in terms of the number of samples. The frame length was calculated by multiplying the frame size (in seconds) by the sampling rate. In this case, the frame length is 1200 samples. The frame step was computed similarly, resulting in a frame step of 480 samples. The total number of frames was calculated to be 1312 based on the signal length and frame parameters.

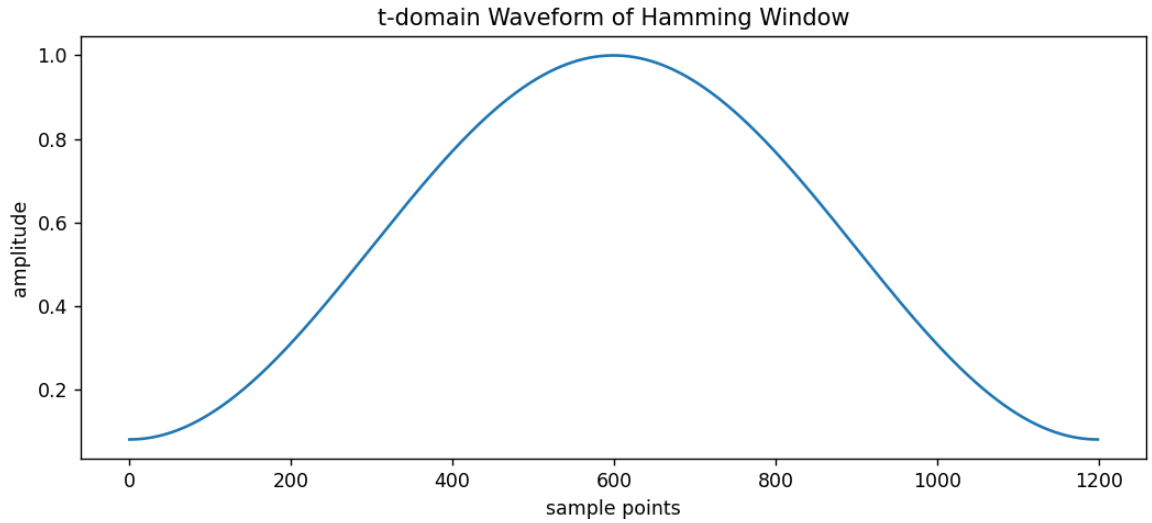
> Frame size: 0.025 seconds
> Frame stride: 0.010 seconds
> Frame length: 1200 samples
> Frame step: 480 samples
> Total number of frames: 1312

Numpy array `frames` was created to store all the frames. First, we created a Numpy array `pad_signal` to store the signal after zero-padding to ensure that the last frame is complete. Then we generated frame indices using the `indices` variable.

```
# Pad the signal with zeros
pad_signal_length = num_frames * frame_step + frame_length
padded_signal = np.pad(y_preemphasized, (0, pad_signal_length - signal_length),
'constant')
# Create frame indices for extraction
indices = np.tile(np.arange(frame_length), (num_frames, 1)) +
np.tile(np.arange(0, num_frames * frame_step, frame_step), (frame_length, 1)).T
frames = padded_signal[indices]
```

Hamming window was applied to each frame, which helps reduce spectral leakage and prepares the frames for further analysis.

```
frames *= np.hamming(frame_length)
```



Short Time Fourier Transform (STFT)

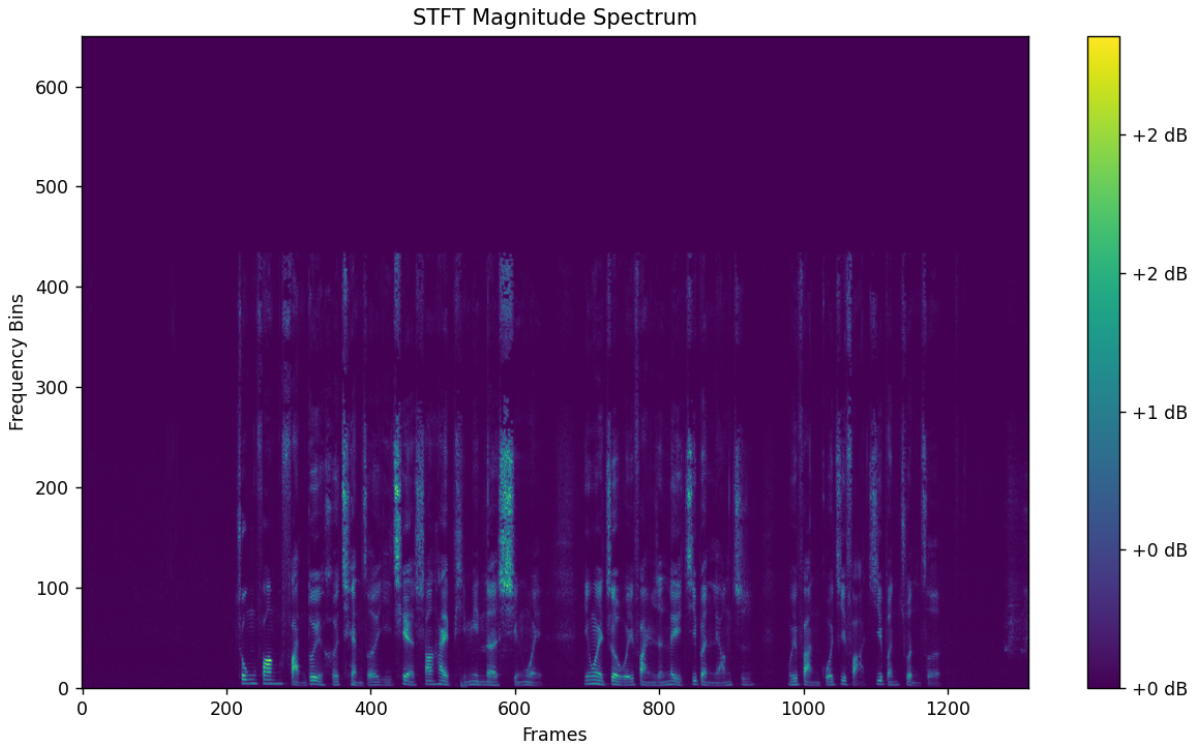
STFT was performed on each frame to transform the time-domain signal into the frequency domain.

```
n_FFT = 1300  # size of FFT
hop_length = frame_step  # time interval between frames

stft_matrix = np.array([np.fft.fft(frame, n=n_FFT) for frame in frames]) #
initialize STFT matrix
stft_matrix = stft_matrix[:, :n_FFT // 2 + 1]    # we only need the positive
portion

# calculate amplitude spectrum of STFT
magnitude = np.abs(stft_matrix)
# calculate energy spectrum
energy = magnitude ** 2
```

The magnitude spectrum and energy spectrum of the STFT were calculated and visualized.



Mel-filter Bank

The Mel-filter bank was applied to weight the energy of audio signals in the Mel scale. For this analysis, the number of filters was set to 40. And to define the Mel filter bank, we needed to convert frequencies between the Mel and Hz scales. We determined the lower and upper limits of the Mel scale in Hz and evenly spaced Mel points between these two limits.

```
n_filter = 40      # number of filters

# dividing Mel scale
low_freq_mel = 0
high_freq_mel = (2595 * np.log10(1 + (sr / 2) / 700))
mel_points = np.linspace(low_freq_mel, high_freq_mel, n_filter + 2)
hz_points = (700 * (10**(mel_points / 2595) - 1))   # convert Mel back to Hz
```

The Mel filter bank was defined as a matrix `fbank`. We iterated through each filter and determined its left and right boundaries by indexing the 'bin' array. And for each filter, we computed the weight for each frequency bin within its boundaries, following the Mel scale. The resulting filter bank matrix contained the weights applied to energy values in the STFT.

```
# Calculate the Mel filter bank
bin = np.floor((n_FFT + 1) * hz_points / sr).astype(int)  # Calculate the bin
index of each Mel filter in FFT

# Initialize the matrix of fbank
fbank_matrix = np.zeros((n_filter, int(np.floor(n_FFT / 2 + 1))))

# Calculate the Mel filter bank weights using vectorized operations
for filt in range(1, n_filter + 1):
    left_boundary = bin[filt - 1]  # Left boundary
    middle = bin[filt]  # Middle
    right_boundary = bin[filt + 1]  # Right boundary

    # Calculate the weight of bin for each frequency using vectorized operations
    w1 = np.arange(left_boundary, middle)
    w2 = np.arange(middle, right_boundary)

    fbank_matrix[filt - 1, w1] = (w1 - bin[filt - 1]) / (bin[filt] - bin[filt -
1])
    fbank_matrix[filt - 1, w2] = 1 - (w2 - bin[filt]) / (bin[filt + 1] -
bin[filt])
```

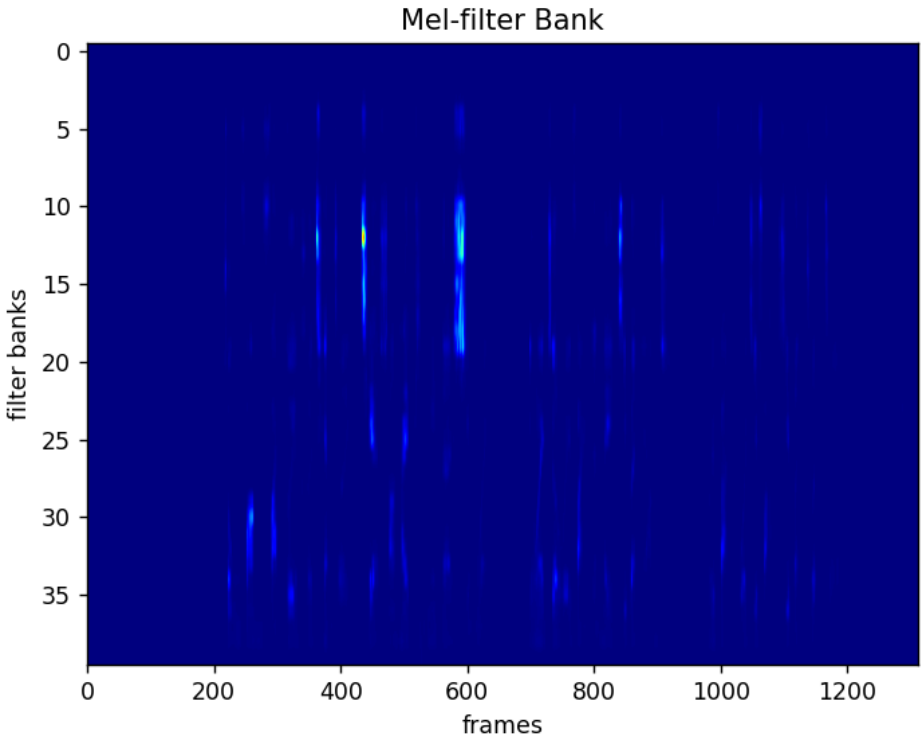
To create the Mel-filter bank features, the energy spectrum values was calculated with the filter bank matrix using matrix multiplication. Noticed that to avoid numerical instability issues when values are close to zero, zero-valued elements were replaced with the machine epsilon.

```
# Weight the energy of audio signals on the Mel scale using matrix multiplication
filter_banks = np.dot(energy, fbank_matrix.T)
# Ensure numerical stability by replacing zeros with epsilon
filter_banks = np.where(filter_banks == 0, np.finfo(float).eps, filter_banks)
```

The Mel-filter bank output matrix was printed, providing a numerical representation of the filter bank's weights for each frame and filter.

> Mel-filter bank output matrix:
> [[2.22044605e-16 2.22044605e-16 2.22044605e-16 ... 2.22044605e-16
> 2.22044605e-16 2.22044605e-16]
> [2.22044605e-16 2.22044605e-16 2.22044605e-16 ... 2.22044605e-16
> 2.22044605e-16 2.22044605e-16]
> [6.14547675e-12 9.44670581e-12 1.16822159e-11 ... 6.37387525e-09
> 3.76848825e-09 1.50421890e-09]
>
> ...
> [9.60308729e-04 5.24108462e-04 7.86435396e-04 ... 9.14376086e-06
> 9.94242459e-06 8.92525498e-06]
> [1.01193260e-03 2.23342681e-04 1.27239392e-03 ... 7.36769421e-06
> 8.34389680e-06 6.88335547e-06]

> [1.93432146e-03 2.57915932e-03 9.66372047e-04 ... 5.26117414e-06
> 5.89197784e-06 6.84276415e-06]]



Log Transformation

Aiding in data compression and pattern recognition, a logarithmic transformation (in dB) was applied to the Mel-filter bank output.

```
filter_banks = 20 * np.log10(filter_banks)  # dB
```

Discrete Cosine Transform (DCT)

In speech processing, DCT is usually used to extract MFCC features, which encode the spectral characteristics of audio signals for speech recognition and audio processing. `dct` function imported from `scipy.fftpack` was used in this step to reduce data dimensionality while retaining critical features.

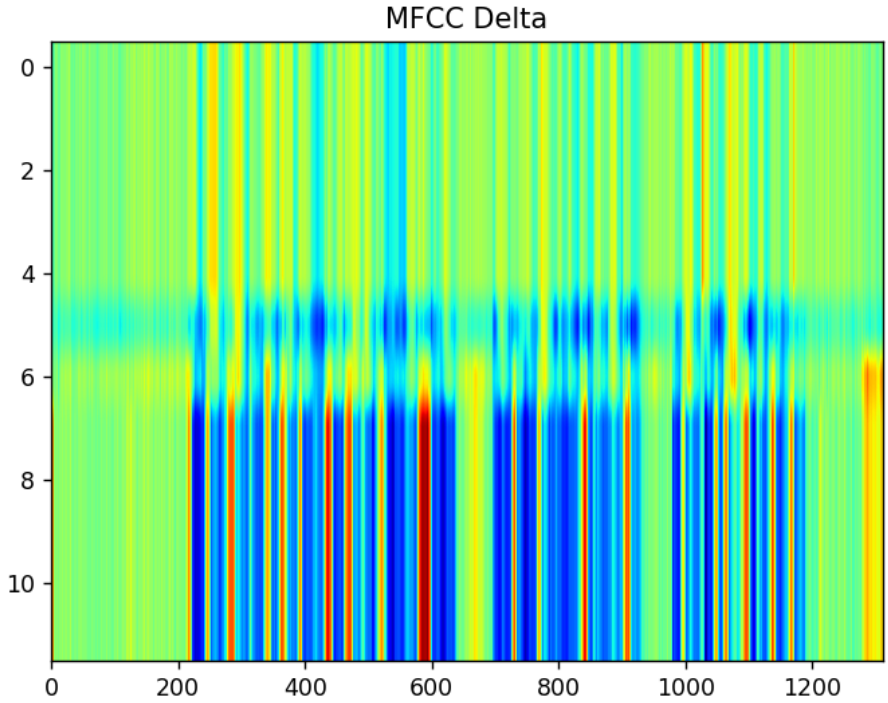
```
num_coefficients = 12   # hyperparameter: number of coefficients to retain
mfcc = dct(filter_banks, type=2, axis=1, norm='ortho')[:, 1: (num_coefficients + 1)]
```

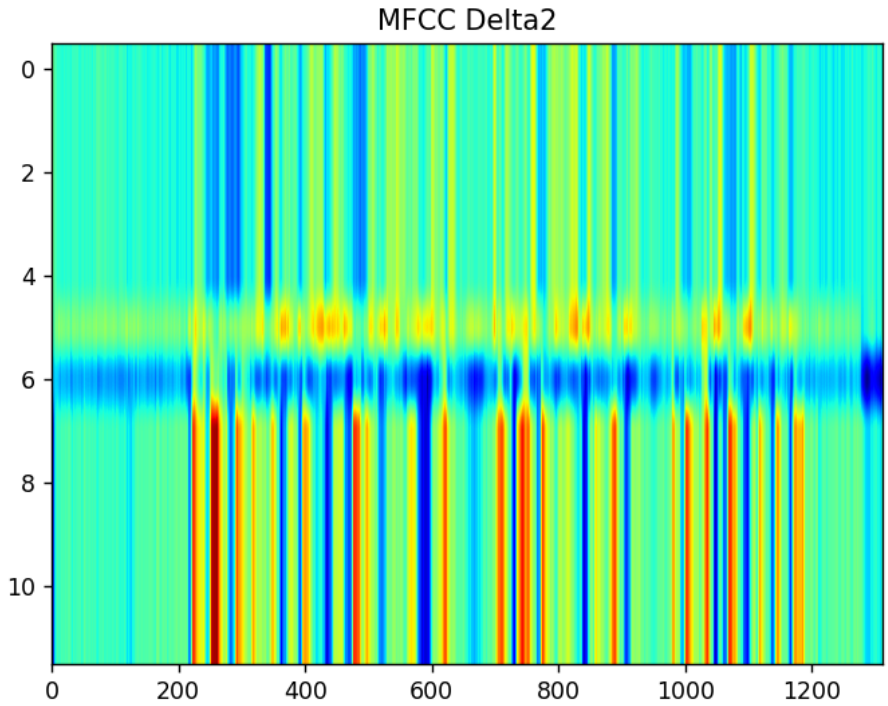
Dynamic Feature Extraction

We computed the first and second-order temporal derivatives (MFCC Delta and Delta2) to capture dynamic changes in the audio signal. These dynamic features could enhance the robustness of MFCCs for ASR.

```
mfcc_delta = librosa.feature.delta(mfcc)
mfcc_delta2 = librosa.feature.delta(mfcc, order=2)
```

Then we get the matrices of MFCC Delta and Delta2 (see the file `mfcc_delta.xlsx` and `mfcc_delta2.xlsx`)





Feature Transformation

Feature transformation was conducted, including using cepstral mean normalization (CMN) and cepstral variance normalization (CVN), to enhance the stability and consistency of MFCCs. CMN helps make the features robust to linear filtering of the signal, such as variations caused by different channels, while CVN is a process that aims to normalize the variance of each feature vector element to 1.

```
# CMN
mean = np.mean(mfcc, axis=0)
mfcc -= mean

# CVN
std = np.std(mfcc, axis=0)
mfcc /= (std + 1e-8)  # to avoid dividing by 0
```

Outcome of MFCC process

The MFCC matrix, after feature transformation, was printed to the console as well as saved in Excel files (see the file `final_mfcc.xlsx`).

> MFCC matrix:
> [[-0.75864 1.75434 -1.84861 ... 1.7002 -1.32654 -0.13141]
> [-0.75864 1.75434 -1.84861 ... 1.7002 -1.32654 -0.13141]
> [-2.43782 1.05072 -1.77653 ... 0.56487 -1.96935 -0.30418]
> ...
> [-0.43704 -2.25363 -1.02883 ... 0.76998 -2.08728 1.514]
> [-0.48058 -2.22954 -1.24998 ... 0.50946 -1.32079 0.6798]
> [-0.45802 -1.60597 -0.51177 ... 0.77663 -1.23785 0.79887]]

We also visualized the transformed MFCCs using a heatmap representation, which provides a clear illustration of how the MFCC coefficients vary over time and across different feature dimensions.