VAD（频谱熵）算法报告

算法整体思路与实现细节

通过频谱熵来检测音频信号中的语音活动。频谱熵用于衡量频谱信息的复杂度，用于区分语音信号和非语音信号。算法主要步骤如下：

1. **信号预处理**：音频信号在处理前进行去均值和归一化，以减小噪声影响。
2. **信号分帧**：将音频信号分割成短时间帧，每帧包含固定数量的采样点。这使得处理可以在短时间内进行，从而更精确地检测语音活动。实现该步骤的函数是 `enframe`，其作用是将输入信号 `x` 分割成帧。

```
def enframe(x, win, inc):
    nx = len(x)
    if isinstance(win, list) or isinstance(win, np.ndarray):
        nwin = len(win)
        nlen = nwin  # Frame length = window length
    elif isinstance(win, int):
        nwin = 1
        nlen = win  # Set to frame length
    if inc is None:
        inc = nlen
    nf = (nx - nlen + inc) // inc
    frameout = np.zeros((nf, nlen))
    indf = np.multiply(inc, np.array([i for i in range(nf)]))
    for i in range(nf):
        frameout[i, :] = x[indf[i]:indf[i] + nlen]
    if isinstance(win, list) or isinstance(win, np.ndarray):
        frameout = np.multiply(frameout, np.array(win))
    return frameout
```

3. **滑动窗口**：使用汉明窗对信号进行加窗处理，减小频谱泄漏。
4. **计算频谱**：对每一帧进行快速傅里叶变换（FFT），获取频谱信息。然后计算每帧频谱的概率分布，并基于此计算熵值。频谱熵值低表示频谱信息集中（可能是语音信号），频谱熵值高表示频谱信息分散（可能是噪音或静音）。
5. **设定阈值**：根据初始静音帧的频谱熵值设定两个阈值，用于区分语音和非语音信号。实现该步骤的函数是 `vad_specEN`，其作用是基于频谱熵进行语音活动检测。

```
def vad_specEN(data, wnd, inc, NIS, thr1, thr2, fs):
    x = enframe(data, wnd, inc)
    X = np.abs(np.fft.fft(x, axis=1))
    if len(wnd) == 1:
        wlen = wnd
    else:
        wlen = len(wnd)
    df = fs / wlen
    fx1 = int(250 // df + 1)
    fx2 = int(3500 // df + 1)
    K = 0.5
    E = np.zeros((X.shape[0], wlen // 2))
    E[:, fx1 + 1:fx2 - 1] = X[:, fx1 + 1:fx2 - 1]
```

```
    E = np.multiply(E, E)
    Esum = np.sum(E, axis=1, keepdims=True)
    P1 = np.divide(E, Esum)
    E = np.where(P1 >= 0.9, 0, E)
    Eb0 = E[:, 0::4]
    Eb1 = E[:, 1::4]
    Eb2 = E[:, 2::4]
    Eb3 = E[:, 3::4]
    Eb = Eb0 + Eb1 + Eb2 + Eb3
    prob = np.divide(Eb + K, np.sum(Eb + K, axis=1, keepdims=True))
    Hb = -np.sum(np.multiply(prob, np.log10(prob + 1e-10)), axis=1)
    Hb = medfilt(Hb, 5)
    Me = np.mean(Hb)
    eth = np.mean(Hb[:NIS])
    Det = eth - Me
    T1 = thr1 * Det + Me
    T2 = thr2 * Det + Me
    voiceseg = vad_revr(Hb, T1, T2)
    return voiceseg
```

6. **检测语音段**：通过比较每一帧的频谱熵值与设定的阈值，检测并标记语音活动段。实现该步骤的函数是 `vad_revr` 和 `findSegment`。

```
def vad_revr(dst1, T1, T2):
    fn = len(dst1)
    maxsilence = 8
    minlen = 5
    status = 0
    count = np.zeros(fn)
    silence = np.zeros(fn)
    xn = 0
    x1 = np.zeros(fn)
    x2 = np.zeros(fn)
    for n in range(1, fn):
        if status == 0 or status == 1:
            if dst1[n] < T2:
                x1[xn] = max(1, n - count[xn] - 1)
                status = 2
                silence[xn] = 0
                count[xn] += 1
            elif dst1[n] < T1:
                status = 1
                count[xn] += 1
            else:
                status = 0
                count[xn] = 0
                x1[xn] = 0
                x2[xn] = 0
        if status == 2:
            if dst1[n] < T1:
                count[xn] += 1
            else:
                silence[xn] += 1
                if silence[xn] < maxsilence:
                    count[xn] += 1
                elif count[xn] < minlen:
                    status = 0
```

```
                    silence[xn] = 0
                    count[xn] = 0
                else:
                    status = 3
                    x2[xn] = x1[xn] + count[xn]
        if status == 3:
            status = 0
            xn += 1
            count[xn] = 0
            silence[xn] = 0
            x1[xn] = 0
            x2[xn] = 0
    el = len(x1[:xn])
    if x1[el - 1] == 0:
        el -= 1
    if x2[el - 1] == 0:
        print('Error: Not find ending point!\n')
        x2[el] = fn
    SF = np.zeros(fn)
    for i in range(el):
        SF[int(x1[i]):int(x2[i])] = 1
    voiceseg = findSegment(np.where(SF == 1)[0])
    return voiceseg

def findSegment(express):
    if express[0] == 0:
        voiceIndex = np.where(express)
    else:
        voiceIndex = express
    d_voice = np.where(np.diff(voiceIndex) > 1)[0]
    voiceseg = {}
    if len(d_voice) > 0:
        for i in range(len(d_voice) + 1):
            seg = {}
            if i == 0:
                st = voiceIndex[0]
                en = voiceIndex[d_voice[i]]
            elif i == len(d_voice):
                st = voiceIndex[d_voice[i - 1] + 1]
                en = voiceIndex[-1]
            else:
                st = voiceIndex[d_voice[i - 1] + 1]
                en = voiceIndex[d_voice[i]]
            seg['start'] = st
            seg['end'] = en
            seg['duration'] = en - st + 1
            voiceseg[i] = seg
    return voiceseg
```

主程序流程：

1. **加载音频文件**：使用 `librosa` 库加载音频文件，并进行预处理。
2. **调用VAD函数**：使用 `vad_specEN` 函数检测语音活动。
3. **保存结果**：将检测结果保存到指定文件。

```
def vad(wav):
    wav_input, sample_rate = librosa.load(wav, sr=SAMPLE_RATE)
    wav_input = wav_input - np.mean(wav_input)
    wav_input /= np.max(np.abs(wav_input))
    wnd = np.hamming(FRAME_LENGTH)
    voiceseg = vad_specEN(wav_input, wnd, HOP_LENGTH, NIS, THR1, THR2,
sample_rate)
    vad_output = [[seg['start'] * HOP_LENGTH, seg['end'] * HOP_LENGTH] for seg
in voiceseg.values()]
    return vad_output

if __name__ == '__main__':
    vad_output = vad(WAV)
    print(vad_output)
    with open(PREDICT_TXT_SAVE_PATH, "w") as f:
        for i in range(len(vad_output)):
            f.write(str(vad_output[i][0]))
            f.write(',')
            f.write(str(vad_output[i][1]))
            f.write('\n')
```

测试结果

data_1.wav

f1_score: 0.9501407045205069
accuracy: 0.9750464571221653
recall: 0.9180222916425623
precision: 0.9845880211780278

data_2.wav

f1_score: 0.9083884049264044
accuracy: 0.9566424491272112
recall: 0.9612285441830897
precision: 0.8610549544419135

data_3.wav

f1_score: 0.9165144374144807
accuracy: 0.9283538691674126
recall: 0.9226976240374879
precision: 0.9104135687732342

####

算法C代码实现

`vad.h`

这个头文件定义了VAD算法的接口，以及一些常量和类型。

`vad.c`

这个源文件实现了谱熵法的VAD算法。

- 实现了信号的分帧处理函数`enframe`。
- 实现了谱熵计算函数`calculate_spectral_entropy`。

- 实现了反向比较函数`vad_revr`，用于确定语音活动的起始和结束点。
- 实现了主处理函数`vad_proc`，用于调用上述函数完成VAD过程。

`main.c`

调用VAD算法并输出结果

`test_data.h`

这个头文件定义了一个包含测试数据的数组。

测试结果

```
Vocal signal startpoint and endpoint:
160 1760
2160 4480
9040 9760
11920 12800
13760 16000
16160 17920
18080 19920
20160 21680
21920 22640
27840 29280
29520 30400
30720 33040
33760 34640
35440 36080
36320 38400
38640 39600
41360 42000
45760 47120
47280 48240
48800 51040
51520 53440
53840 54480
54640 55280
55520 56240
62160 63360
63520 64240
64960 66960
67120 69280
69680 70720
70880 71520
71680 72320
73200 73920
76960 77600
79600 80560
80720 82320
82480 83600
83760 84800
87280 88720
88880 89760
94080 94720
```