Project 1：语音端点检测

519030910349 薛峥嵘

1. 基于线性分类器和语音短时能量的简单语音端点检测算法

1.1. 数据预处理及特征提取

1.1.1. 数据预处理

首先，对于读入的原始语音数据进行去直流操作和归一化操作。直流分量一般是由录音仪器导致的，它对会信号的能量判断造成干扰。为去除直流分量，我们只需将语音数据逐点减去平均值即可。随后进行信号的归一化。归一化可以确保语音数据逐点位于 [-1,1] 区间内，这样做是为了使 VAD 检测在面对不同响度的语音信号时更加鲁棒。

其次，进行分帧操作。由于语音信号存在较强的上下文关联，我们在分帧时需保证相邻的两帧之间包含一定的重叠部分。在这个项目中，我们令每一帧包含 512 个点，每次帧移的距离是 128 个点。

数据预处理的伪代码如下。

```
def pre_processing(raw_data, frame_size, frame_shift):
    raw_data = raw_data - mean(raw_data)
    data = raw_data / max(abs(raw_data))
    frame_count = (len(data) - frame_size + frame_shift) / frame_shift
    frame_list = []
    for i in range(frame_count):
        frame_list.append(data[i * frame_shift : (i+1) * frame_shift])
    return frame_list
```

1.1.2. 特征提取

在本次实验中，我们提取了两种特征：短时能量和短时过零率。

一个语音片段的短时能量越小，它越有可能是一个静音的片段，反之则更有可能是人声的片段。这是十分符合直观的。更加具体地说，无声段或背景噪声段能量最低；浊音段是声带振动段，能量最高；清音段是空气在口腔中的摩擦、冲击或爆破而发出的语音信号段，能量居中。

计算短时能量伪代码如下。

```
def get_energy(wav_slice):
    wav_square = wav_slice * wav_slice
    energy = sum(wav_square)
    return energy / len(wav_slice)
```

另外，短时过零率也是一个十分重要的指标。短时过零率是一帧语音信号中改变符号的比例。浊音段频率较低，有较低的过零率；清音段频率较高，有较高的过零率。

计算短时过零率的伪代码如下。

```
def get_czr(wav_slice):
    count = 0
    for i in range(len(wav_slice)-1):
        if wav_slice[i] * wav_slice[i+1] < 0:
            count += 1
    return count / len(wav_slice)
```

特征提取的有关代码被打包进了 features.py 这个文件中。

1.2. 算法描述

针对短时过零率和短时能量，我们分别手动设置它们的高阈值和低阈值。直观上，如果某一帧的特征高于高阈值，那么它有很高概率是人声的状态；如果某一帧的特征低于低阈值，那么它有很高概率是静音状态；如果介于高低阈值之间，则难以确定。

考虑到人类的发声是一个连续的过程，我们在这里采用一个双门限的方法。如果当前帧被判定为人声状态，那么它的下一帧将维持人声状态，除非下一帧的短时能量和短时过零率同时低于低阈值。同理，如果当前帧被判定为静音状态，那么它的下一帧将维持静音状态，除非下一帧的短时能量和短时过零率同时高于高阈值。

利用连续发声的特性，我们还可以对输入的特征以及输出的预测标签进行平滑滤波，同时删除持续时间特别小的人声片段，来进一步提升模型的性能。

算法的伪代码如下。输入是已经完成分帧操作的能量和过零率序列，输出是一个逐帧的预测标签。

```
def main_algorithm(czr_list, energy_list, filter_length = 32):
    labels = []
    for j in range(len(czr_list)):
        # a little smoothing on input
        czr_list[j] = mean(czr_list[j : j+3])
        energy_list[j] = mean(energy_list[j : j+3])

        # predict label depending on last frame's label
        if last_label == 1:
            if czr_list[j] < low_threshold_czr and energy_list[j] < low_threshold_energy:
                predict_label = 0
            else:
                predict_label = 1
        if last_label == 0:
            if czr_list[j] > high_threshold_czr and energy_list[j] > high_threshold_energy:
                predict_label = 1
            else:
                predict_label = 0
        labels.append(predict_label)
        last_label = predict_label

    # smoothing on output labels
    for j in range(len(labels)):
        labels = mean(labels[j : (j+filter_length)])

    # delete some too short speech
    count = 0
    for j in range(len(labels)):
        if label[j] == 1:
            count += 1
        if label[j] == 0 and 7 >= count >= 1:
            labels[j-count : j] = 0
        if label[j] == 0:
            count = 0
```

1.3. 实验结果

我们采用准确率、AUC、ERR 这三个指标来评价我们的模型的性能表现。准确率就是逐帧地判断预测标签和真实标签是否一致，并计算相同数和总数的比值。AUC 和 ERR 的计算方法已经在 evaluate.py 中给出，我们简单地调用就可以了。

调参的过程是在开发集上进行的，我们主要调整短时能量和短时过零率的高低阈值。我们给出一组相对最优的超参数和对应的在开发集上的性能表现。

Table 1: Optimal Hyper Parameters

high czr	low czr	high energy	low energy
0.43	0.37	0.0005	0.0002

Table 2: Performance on Development Set

Accuracy	AUC	ERR
0.931	0.933	0.070

为了验证模型的鲁棒性，我们用相同的参数在规模更大且模型没有见过的训练集上进行测试，测试的结果如下表。

Table 3: Performance on Training Set

Accuracy	AUC	ERR
0.931	0.936	0.073

模型在陌生的数据集上的性能表现与之前基本一致，这表明我们的模型有较好的鲁棒性。

2. 基于统计模型分类器和语音频域特征的语音端点检测算法

2.1. 数据预处理及特征提取

2.1.1. 数据预处理

与任务 1 一样，我们对原始的语音信号进行了去直流操作和归一化操作，在此不再赘述。

2.1.2. 特征提取

在任务 2 中，我们提取了语音信号的 MFCC 特征，在这里做一个简单的介绍。

梅尔域 (Mel Scale) 是一个基音感知域，它通过听音者可以区分两种纯音频率的差距来作为标度。人耳对于不同频域的信号的敏感程度是不同的，研究表明它对低频的差异敏感程度高于高频的差异，因此，梅尔域和线性频域之间通过一个非线性的映射函数相互转换，这个映射函数是基于实验经验的拟合结果。

Filter bank(FBank) 系数是一系列带通滤波器 (一般用三角窗滤波器) 分别作用在信号上得到的加权的结果。我们这样做的动机是，短时傅里叶变换得到的幅值谱包含太多的信息，用加权和替代可以大幅地压缩信息，便于计算。

MFCC 特征则是将 FBank 系数取对数并做离散余弦变换，取消各个系数间的相关性，得到的倒谱系数。MFCC 特征比较好地描述了音频的静态特征，但对于动态特征的描述有所缺失，因此，我们可以加入一阶、二阶差分量来进一步丰富 MFCC 特征的信息。

在实验中，我们调用了 python_speech_features 中的 mfcc 和 delta 函数来提取 MFCC 特征及其查分特征。特征提取的具体实现被打包到了 features.py 这个文件中。需要特别说明的是，我们必须对整段音频调用该函数。如果将音频先分帧再调用函数，则 CPU 始终只能单线程工作，效率很低。

2.2. 算法描述

2.2.1. 高斯混合模型

高斯混合模型 (GMM) 描述多维数据的分布，是多个多维高斯分布分量的加权和。能够训练的参数为：每个高斯分量的占有权重 c_j，均值向量 μ_j，协方差矩阵 Σ_j。模型的训练过程是通过 EM 算法实现的。关于 EM 的推导在之前的作业中已经完成了，在这里，我们简单地调用 sklearn.mixture.GaussianMixture 函数实现高斯混合模型。

在实验中，我们创建了两个高斯混合模型的实例：gmm_voice 和 gmm_silence，分别将标注为语音和静音的帧对应的 MFCC 特征和差分量喂进模型进行训练。训练完成后，将开发集逐帧地输入这两个模型，分别计算似然值。如果 gmm_voice 的似然更大，我们判定这帧对应语音。如果 gmm_silence 的似然更大，我们判定这帧是静音。

2.2.2. 更多实现细节

与任务 1 相比，高斯混合模型的训练需要相当长的时间。如果对整个训练集进行训练，时间开销稍大。我们在实验中发现，当训练的帧数大于十万后，模型性能的提升显著变慢。因此，我们在实验中选取了前一百万帧进行训练。

高斯混合模型的高斯分量的数量是需要调整的超参数。实验表明，当数量较小时，模型的拟合性能较差；当数量较大时，模型的训练时间过长，且出现了一定的过拟合现象。经过一些尝试后，我们选取 10 个高斯分量。

与任务 1 一样，我们利用发声的连续性特征进行平滑处理。在数据后处理时，我们对 MFCC 特征进行平滑处理。同时，删去时长过短的语音片段。

2.2.3. 实验结果

在这里，我们展示三种实现方式的结果：只使用 MFCC 特征、MFCC 特征 + 一阶差分量、MFCC 特征 + 一阶差分量 + 二阶差分量。

Table 4: Performance given different settings

Setting	Accuracy	AUC	ERR
MFCC only	0.944	0.942	0.058
MFCC + 一阶差分	0.948	0.949	0.052
MFCC + 一二阶差分	0.950	0.949	0.053

可以看到，相较于任务 1，任务 2 的方法有了显著的性能提升。加入差分量可以进一步提升性能，但性能提升是比较有限的。