项目文档

> 学号：1952230
>
> 姓名：朱增乐
>
> 指导教师：沈莹

1. 目录结构

```
├── mfcc_samples
|   ├── 各个音频mfcc的二进制文件
├── speech_feature
|   ├── mfcc.py
|   └── process.py
├── wav
|   ├── 音频文件
├── hmm.py
├── hmm_utils.py
├── main.py
├── mfcc_utils.py
├── testing_file.csv
├── training_file.csv
├── main.py
```

2. 所需要的包

- Python 3.8.5
- numpy 1.19.2
- matplotlib 3.3.2
- scipy 1.5.2
- progressbar2 3.55.0

3. 运行方法

1. 安装所必须的包，请注意`progressbar`的安装指令，需要输入的是`pip install progressbar2`，`progressbar2`能够在IDEA运行时正常使用，而progressbar无法在PyCharm中正常使用。需要注意的是，这里为了加速运行，使用了多线程。在本机电脑上的极限线程数大概是15个，因此设立的并行执行数也是15个。如果运行时计算机性能较好，则可考虑增加并行执行数，如果性能较弱，则需减少并行执行数。

2. 在运行过程中，matplotlib可能无法正常展示出曲线，这个是正常现象，不影响程序执行。在imgs文件夹下保存有运行时的图片文件。在整个程序执行完毕后，曲线应当可以正常展示。这个现象应当是由matplotlib包中自身实现所导致的。
3. 运行方法

   （1）PyCharm（推荐）

   直接运行main.py

   （2）命令行

   在main.py的同级目录下，输入指令`python main.py`即可运行。我设置了一些命令行参数，其基本作用如下：

参数名	作用
$start$	状态数的起始个数，为了测试不同状态数的预测分类效果，将使用[start, end]依次作为状态的个数，对模型进行训练，默认值为12
end	状态数的终止个数，默认值为14
dim	mfcc维度，默认值为39
$model$	总类别个数，默认值为11
$flag$	是否需要重新生成mfc文件（mfcc的二进制文件），默认值为true
lib_flag	是否调用mfcc库函数，默认值为false

4. 主要函数介绍

4.1 mfcc相关函数(speech_feature下的函数)

4.1.1 pre_emphasis

- 函数定义

```
def pre_emphasis(signal, alpha=0.97):
```

- 参数

```
signal: 输入信号
alpha: 默认 0.97 通常为 0.96~0.98
```

- 作用

```
将信号进行预加重
```

- 返回值

```
return: 预加重后的信号
```

4.1.2 windowing

- 函数定义

```
def windowing(signal, rate, frame_size_sec=0.025, frame_step_sec=0.01,
window_name="hamming"):
```

- 参数

```
signal: 输入信号
rate: 采样率
frame_size_sec: 每个窗口分析时长
frame_step_sec: 窗口每次增长时长
window_name: 加窗函数名
```

- 作用

```
对信号进行加窗处理，窗口默认使用hamming窗，同时也提供blackman窗和bartlett窗。
```

- 返回值

```
return: 加窗后的信号
```

4.1.3 dft

- 函数定义

```
def dft(signal, nfft):
```

- 参数

```
signal: 输入信号
nfft: n维变换
```

- 作用

```
对信号进行傅里叶变换
```

- 返回值

```
return: 经过傅里叶变换后的信号
```

4.1.4 mel_filter

- 函数定义

```
def mel_filter(nfft, pow_frames, rate, filter_banks=26):
```

- 参数

```
nfft: n维变换
pow_frames: 平方后的帧
rate: 采样率
filter_banks: 滤波器组
```

- 作用

```
对信号进行mel滤波处理。
```

- 返回值

```
return: 经过mel滤波后的信号
```

4.1.5 signal_filter

- 函数定义

```
def signal_filter(signal, rate, **kwargs):
```

- 参数

```
signal: 原始信号
rate: 采样率
kwargs: 关键字参数 包括alpha 窗口长度 窗口每次增长长度 滤波器组数（4.1.1~4.1.4函数中的参数）等等
```

- 作用

```
调用pre_emphasis、windowing、dft和mel_filter对信号进行滤波处理。同时对计算所获得的特征及能量进行log运算，保证处于同一数量级，最终返回特征和能量两个矩阵。
```

- 返回值

```
return: feature 特征
        energy  能量
```

4.1.6 dfe

- 函数定义

```
def dfe(feature, energy, cep_lifter=22):
```

- 参数

```
feature: 特征
energy: 能量
cep_lifter: 提升器
```

- 作用

```
dynamic feature extraction， 梅尔频率倒谱系数2~13维特征保留，第一维替换为能量。
```

- 返回值

```
return: 提取后的特征（能量替换值第一维）
```

4.1.7 delta_feature

- 函数定义

```
def delta_feature(feature, n):
```

- 参数

```
feature: mfcc特征
n: 差分阶数
```

- 作用

```
计算n阶差分后的mfcc特征。
```

- 返回值

```
return: n阶差分后的mfcc特征
```

4.1.8 mfcc

- 函数定义

```
def mfcc(signal, rate, **kwargs):
```

- 参数

```
signal: 原始信号
rate: 采样率
kwargs: 关键字参数 包括特征提取数 倒谱升降器数量 差分阶数等等
      通过向该关键字参数赋值 可以获得不同的mfcc特征
```

- 作用

```
计算mfcc。
```

- 返回值

```
return: mfcc
```

4.2 universal_utils.py

4.2.1 generate_training_list

- 函数定义

```
def generate_training_list():
```

- 参数

```
无
```

- 作用

```
选择'AE', 'AJ', 'AL', 'AW', 'BD', 'CB', 'CF', 'CR', 'DL', 'DN', 'EH',
'EL', 'FC', 'FD', 'FF', 'FI', 'FJ', 'FK', 'FL', 'GG'文件夹下的所有音频数据
作为训练数据，并将其文件路径以及所对应的文字写入training_file.csv文件内。
```

- 返回值

```
无
```

4.2.2 generate_testing_list

- 函数定义

```
def generate_testing_list():
```

- 参数

无

- 作用

和generate_training_list类似，该函数无参数。它是选择'AH', 'AR', 'AT', 'BC', 'BE', 'BM', 'BN', 'CC', 'CE', 'CP', 'DF', 'DJ', 'ED', 'EF', 'ET', 'FA', 'FG', 'FH', 'FM', 'FP', 'FR', 'FS', 'FT', 'GA', 'GP', 'GS', 'GW', 'HC', 'HJ', 'HM', 'HR', 'IA', 'IB', 'IM', 'IP', 'JA', 'JH', 'KA', 'KE', 'KG', 'LE', 'LG', 'MI', 'NL', 'NP', 'NT', 'PC', 'PG', 'PH', 'PR', 'RK', 'SA', 'SL', 'SR', 'SW', 'TC'文件夹下的所有音频数据作为测试数据，并将其文件路径以及所对应的文字写入testing_file.csv文件内。

- 返回值

无

4.2.3 write_binary_file

- 函数定义

```
def write_binary_file(filename, feature, frame_step_sec):
```

- 参数

```
filename: 文件名
feature: mfcc特征
frame_step_sec: 这个在matlab代码中并没有用到 但为保持一致性 将其保留
```

- 作用

以二进制格式把mfcc写入.mfc文件中

- 返回值

无

4.2.4 read_binary_file

- 函数定义

```
def read_binary_file(filename):
```

- 参数

```
filename: 文件名
```

- 作用

```
write_binary_file的逆过程，以二进制格式从文件中读取mfcc
```

- 返回值

```
return: mfcc特征。
```

4.2.5 get_data

- 函数定义

```
def get_data(filename):
```

- 参数

```
filename: 训练数据csv或测试数据csv文件名
```

- 作用

```
该函数为generate_training_list和generate_testing_list的逆过程。它的参数为filename，表示读取文件的名字。该函数从filename文件中mfc文件所处路径以及对应的文本，存入列表中。列表长度即为训练或测试数据的长度，列表中的数据位一个个的小列表，每个小列表中储存文本及路径。
```

- 返回值

```
return: 从csv中读取一个列表，每一个列表的元素也是一个列表，有类别和文件路径组成
```

4.3 mfcc_utils.py

4.3.1 generate_mfcc_samples

- 函数定义

```
def generate_mfcc_samples(**kwargs):
```

- 参数

```
该函数参数使用关键字参数，其中包括五个参数：
indir:              音频文件所在文件夹，默认值为"wav"
in_filter           正则表达式，用于匹配".w(W)a(A)v(V)"后缀，  默认值为"\.[Ww][Aa][Vv]"
outdir              输出文件夹，默认值为"mfcc_samples"
out_ext             文件后缀名，默认值为".mfc"
outfile_format      输出文件格式，默认值为".htk"
```

- 作用

该函数以递归方式遍历indir文件夹下的音频文件，如果遍历到了文件夹，则在outdir创建一个相同名字的文件夹。如果遍历到了文件，则先用正则校验是否为.wav格式的文件。如果是则的调用fwav2mfcc函数，生成相应的.mfc文件。

- 返回值

无

4.3.2 fwav2mfcc

- 函数定义

```
def fwav2mfcc(infile_name, outfile_name, outfile_format, **kwargs):
```

- 参数

该函数有三个参数以及一个关键字参数，关键字参数用于mfcc各个参数的赋值，这里就不予赘述了
infile_name:　　输入的文件名
outfile_name:　　输出的文件名
outfile_format:　　输出的文件格式
kwargs: 包含mfcc的各种参数

- 作用

该函数的参数在generate_mfcc_samples函数中传入。它会读取infile_name文件，通过mfcc函数计算mfcc特征，并以二进制的格式将mfcc写入outfile_name文件中。不过这只有在outfile_format是htk时才会发生。

- 返回值

无

4.4 hmm.py(HMM类)

4.4.1 __init__

- 函数定义

```
def __init__(self, training_file="training_file.csv",
testing_file="testing_file.csv", dim=39,
             num_of_model=11, num_of_state=12):
```

- 参数

training_file:	训练数据所存放的csv文件，默认为"training_file.csv"
testing_file	测试数据所存放的csv文件，默认为"training_file.csv"
dim	mfcc频谱的维度，默认39,13+13+13
num_of_model	音频数据类别，包括0~9，o共计11种
num_of_state	提取的状态个数

- 作用

该函数为HMM的构造函数，在构造函数中主要起到赋值作用。并为特征均值矩阵、特征方差矩阵，和Aij为状态转移概率矩阵。特征均值矩阵和特征方差矩阵的shape均为（mfcc频谱的维度，状态数，类别数）。
Aij的shape为（状态数+2，状态数+2，类别数）。加2是因为添加了起始和结束状态。

- 返回值

无

4.4.2 EM_initialization_model

- 函数定义

```
def EM_initialization_model(self):
```

- 参数

无

- 作用

该函数为模型的初始化函数，无参数。该函数的作用是为了计算训练数据的均值与方差做准备工作。
在该函数中，会首先根据dim成员变量生成两个(dim,)维度的矩阵，其中一个矩阵用于记录每个训练数据特征和。具体来讲，将(dim, frame_no)维度的mfcc频谱按行方向求和变成(dim,)向量，再把该向量与特征和向量相加。另一个矩阵记录训练数据的特征平方和，和特征和类似，不过按行方向求和后需要再平方才能相加。随后，该函数会调用
calculate_initial_EM_HMM_items计算均值与方差。

- 返回值

无

4.4.3 calculate_initial_EM_HMM_items

- 函数定义

```
def calculate_initial_EM_HMM_items(self, sum_of_features,
sum_of_features_square, num_of_feature):
```

- 参数

```
sum_of_features: 所有样本的特征和
sum_of_features_square: 所有样本的特征平方和
num_of_feature: 特征个数
```

- 作用

该函数用于计算均值与方差。均值即为 $\frac{sum\_of\_features}{num\_of\_feature}$。

而方差为 $\frac{\Sigma_i(feature_i-mean)^2}{num\_of\_feature}=\frac{\Sigma_i feature_i^2-2*mean*\Sigma_i feature+\Sigma_i mean^2}{num\_of\_feature}=\frac{sum\_of\_features\_square}{num\_of\_feature}-mean^2$

最后的公式也就是代码实现时所采用的公式。

Aij设定为从前一个状态转移到下一个状态概率为0.4，而从当前状态转移到当前状态概率为0.6。而从初态到第一个状态的概率为1。

- 返回值

```
无
```

4.4.4 train_model

- 函数定义

```
def train_model(self):
```

- 参数

```
无
```

- 作用

该函数的作用为训练模型，其核心函数在于`fit`函数。在该函数中，`num_of_iteration`规定了训练轮数，每一轮都会调用`fit`函数进行训练。在训练完成后将每一轮训练的最大似然值的对数绘制出来。

- 返回值

```
无
```

4.4.5 fit

- 函数定义

```
def fit(self, train_data, it, progress_bar):
```

- 参数

```
train_data: 训练数据
it: 迭代次数
progress_bar: 进度条类
```

- 作用

该函数的作用是训练模型。由于训练时间过长，了解代码运行完成情况会更好。通过it和progress_bar参数，可以依照已经参与训练的数据量和需要进行训练的数据总量计算出训练进度，并在控制台中显示进度条。
该函数的核心实际上在于EM_HMM_FR函数，这个函数在hmm_utils.py文件夹中，在4.5.1部分会介绍这个函数。
该函数和EM_HMM_FR将模型训练分成两个部分：（1）计算模型训练中各个参数的分子 （2）计算模型训练中各个参数的分母
在这个函数中，主要计算参数的分子项为：
sum_mean_numerator 特征平均值和分子项
sum_var_numerator 特征方差和分子项
sum_aij_numerator 状态转移概率和分子项
sum_denominator 上述三个参数的分母项
在fit函数中，其会将EM_HMM_FR函数所计算得到的分子项按照类别依次求和，最后再除以分母项。根据计算得到的值，对 self.mean、 self.var 和 self.Aij进行更新。
其中，fit函数该句代码 self.Aij[self.num_of_state + 1, self.num_of_state + 1, k] = 1 的意思是终态到终态的概率是1，即不允许终态再转向其他状态。

- 返回值

```
return: log_likelihood 最大似然对数值
        likelihood     最大似然值
```

4.4.6 test_model

- 函数定义

```
def test_model(self):
```

- 参数

```
无
```

- 作用

该函数是用于测试准确率。该函数的作用是遍历测试数据的mfcc特征，并对每个mfcc特征遍历模型类别，获得每个类别对应的概率。选择最大概率类别作为预测类别，与正确类别比较，用于计算准确率。该函数的核心在于调用_test函数。viterbi_dist_FR函数，

- 返回值

```
return: 测试数据的分类准确率
```

4.4.6 _test

- 函数定义

```
def _test(self, filename, k):
```

- 参数

```
filename: 文件名
k: 类别
```

- 作用

该函数是为了帮助测试过程中并行执行所写，其核心在于viterbi_dist_FR函数，通过viterbi算法计算得到概率，viterbibi_dist_FR函数在 4.5.5中有详细介绍。

- 返回值

4.5 hmm_utils.py

4.5.1 EM_HMM_FR

- 函数定义

```
def EM_HMM_FR(mean, var, aij, filename):
```

- 参数

```
mean:                特征均值
var:                 特征方差
aij:                 状态转移概率矩阵
filename:            mfcc所在路径
```

- 作用

在该函数中，主要目的在于计算训练时的各个参数的分子和分母。但每个分子分母实际上也是由不同参数计算得到的，即alpha，beta和gamma。具体做法为：mean和var在前后添加一维，这是为了与aij保持一致，添加起始和终止状态。aij在最后的概率设置为1，也就是当到达终止态时则永远保持终止。在该代码中，将原始的隐马尔可夫模型中求解

alpha和beta转化为求解log_alpha和log_beta，这是为了防止下溢，因为alpha和beta都很小，连续相乘很有可能小于计算机所能表示的最小正数。

转化为对数运算后，乘法变为加法，初始化及迭代按照下述公式计算：

$$\alpha_1^j = p(\boldsymbol{x}_1|q_1 = j)p(q_1 = j)$$

`

$$\alpha_t^j = b_j(\boldsymbol{x}_t)\sum_{i=1}^{J}(\alpha_{t-1}^i \cdot a_{ij}), 1 \le j \le J, 2 \le t \le T$$

对数化后，公式实际上转变为：

$$log(\alpha_t^j) = log(b_j(x_t)) + log(\Sigma_{i=1}^{J}(\alpha_{t-1}^i * a_{ij}))$$

$$

其中要注意的是，发射概率$b_j(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{\frac{(x-\mu)^2}{\sigma^2}}$在取对数后为$log(\frac{1}{\sqrt{2\pi}\sigma}) + \frac{(x-\mu)^2}{\sigma^2}$。这里是向量运算，最终还需要求和，该公式由`logGaussian`函数实现。而后半部分则通过`logSumAlpha`函数计算。`calculate_log_alpha`通过调用这两个函数计算log_alpha。

log_beta的计算方法按照下述公式：

$$\beta_T^j = 1 \qquad for\ j = 1, \dots, J$$

$$\beta_{t-1}^j = \sum_{i=1}^{J} a_{ji} b_i(\boldsymbol{x}_t)\beta_t^i \qquad for\ t = T-1, \dots, 1$$

初始化为时间T时，对状态迭代全部赋值为1。

之后按照公式，计算beta。部分由`logSumBeta`计算。bi概率即为发射概率，使用`logGaussian`函数计算。`calculate_log_beta`通过调用这两个函数计算log_alpha。

第三个计算的是ξ，公式如下：

$$\xi_t^{(i,j)} = p(q_t = i, q_{t+1} = j|X)$$
$$= \frac{p(q_t = i, q_{t+1} = j, X)}{p(X)}$$
$$= \frac{\alpha_t^i a_{ij} b_j(\boldsymbol{x}_{t+1}) \beta_{t+1}^j}{\sum_{j=1}^J \alpha_t^j \beta_t^j}$$

ξ通过之前所计算得到得到log_alpha和log_beta计算，计算过程由`calculate_log_Xi`函数实现。

第四个计算的是gamma，公式如下：

$$\gamma_t^j = p(q_t = j|X) = \frac{\alpha_t^j \beta_t^j}{p(X)} = \frac{\alpha_t^j \beta_t^j}{\sum_{j=1}^J \alpha_t^j \beta_t^j}$$

该部分由`calculate_gamma`函数实现，需要注意的是由于之前一直取了对数，而这里需要计算的是gamma，因此最终计算完log_gamma后还需进行指数化。

通过上述参数，即可计算出特征均值矩阵、特征方差矩阵的分母以及状态转移概率矩阵的分子和分母。

- 返回值

```
return: mean_numerator 特征均值矩阵的分母
        var_numerator 方差均值矩阵的分母
        aij_numerator 概率转移矩阵的分母
        denominator 特征求和后矩阵的分母
        log_likelihood 最大似然对数值
        likelihood 最大似然值
```

4.5.2 logSumAlpha

- 函数定义

```
def logSumAlpha(log_alpha_t, aij_j):
```

- 参数

```
aij_j: 转移概率矩阵
log_alpha_t: 上一个时间状态的log_alpha的值
```

- 作用

```
辅助计算log_alpha
```

- 返回值

```
return: log_sum_alpha(log_alpha计算式中的一部分)
```

4.5.3 logSumAlpha

- 函数定义

```
def logSumBeta(aij_i, mean, var, obs, beta_t1):
```

- 参数

```
aij_i: 转移概率矩阵
mean: 特征均值矩阵
var: 特征方差矩阵
obs: 观测值
beta_t1: 之前时间态下的beta值
```

- 作用

```
辅助计算log_beta
```

- 返回值

```
return: log_sum_beta(log_beta计算式中的一部分)
```

4.5.4 logGaussian

- 函数定义

```
def logGaussian(mean_i, var_i, o_i):
```

- 参数

```
mean_i: 特征均值矩阵
var_i: 方差均值矩阵
o_i: 观测值
```

- 作用

```
计算发射概率
```

- 返回值

```
return: 发射概率
```

4.5.5 calculate_log_alpha

- 函数定义

```
def calculate_log_alpha(N, T, aij, mean, var, obs):
```

- 参数

```
N: 特征数
T: 观测值的时间长度
aij: 概率转移矩阵
mean: 特征均值矩阵
var: 特征方差矩阵
obs: 观测值
```

- 作用

```
EM_HMM_FR的辅助函数 计算log_alpha
```

- 返回值

```
return: log_alpha
```

4.5.5 calculate_log_beta

- 函数定义

```
def calculate_log_beta(N, T, aij, mean, var, obs):
```

- 参数

```
N: 特征数
T: 观测值的时间长度
aij: 概率转移矩阵
mean: 特征均值矩阵
var: 特征方差矩阵
obs: 观测值
```

- 作用

```
EM_HMM_FR的辅助函数 计算log_beta
```

- 返回值

```
return: log_beta
```

4.5.5 calculate_log_Xi

- 函数定义

```
def calculate_log_Xi(N, T, aij, mean, var, obs, log_alpha, log_beta):
```

- 参数

```
N: 特征数
T: 观测值的时间长度
aij: 概率转移矩阵
mean: 特征均值矩阵
var: 特征方差矩阵
obs: 观测值
log_alpha: 通过calculate_log_alpha所计算出的值
log_beta: 通过calculate_log_beta所计算出的值
```

- 作用

```
EM_HMM_FR的辅助函数 计算log_Xi
```

- 返回值

```
return: log_Xi
```

4.5.5 calculate_gamma

- 函数定义

```
def calculate_gamma(N, T, log_alpha, log_beta):
```

- 参数

```
N: 特征数
T: 观测值的时间长度
log_alpha: 通过calculate_log_alpha所计算出的值
log_beta: 通过calculate_log_beta所计算出的值
```

- 作用

```
EM_HMM_FR的辅助函数 计算gamma
```

- 返回值

```
return: log_Xi
```

4.5.5 viterbi_dist_FR

- 函数定义

```
def viterbi_dist_FR(mean, var, aij, obs):
```

- 参数

```
mean: 特征均值矩阵
var: 特征方差矩阵
aij: 概率转移矩阵
obs: 观测值
```

- 作用

在训练完成后，获得参数是不够的，还需要根据这些参数将音频信号分类。viterbi算法是解决如何寻找与给定观察值序列队形的最佳状态序列的问题。该函数的作用就是实现viterbi算法，从而计算每个类别的概率。

由于在HMM中，后一个状态的概率只与前一个状态有关系，因此在已知状态转移概率、发射概率和初始状态的情况下，就可以根据递推公式：

$$V_t(j) = b_j(x_t)max_{1\leq i\leq N}V_{t-1}(i)a_{ij}$$

V：状态概率

b：发射概率

a_{ij}：状态转移概率

计算出每个状态下的概率，一直计算到终态，即可得到一个最优状态和最大概率。不过实际上，我们的目的是对音频分类，对最优状态是不感兴趣的，这可能也是matlab代码中保留了状态转移过程但是并没有使用的原因。通过计算不同类别下的最大概率，即可做到对音频的预测分类。

其算法流程如下：

（1）初始化：

$$V_1^j = \frac{b_j(x_1)}{\#states}$$

不过，这里采用的都是对数运算，所以这里实际上也进行了对数化。

（2）迭代：

$$V_t^j = b_j(x_t) \max_i(V_{t-1}^i * a_{ij}) \text{ for all states } i, j \in S, t \geq 2.$$

- 返回值

```
return: 当前类别下的最大概率
```

5. 性能分析

5.1 Python运行截图

说明，由于python运行hmm时过慢，因此加入了并行优化。不过，为了展现并行优化的效果，我将之前没有加入并行优化的截图也放了上来。

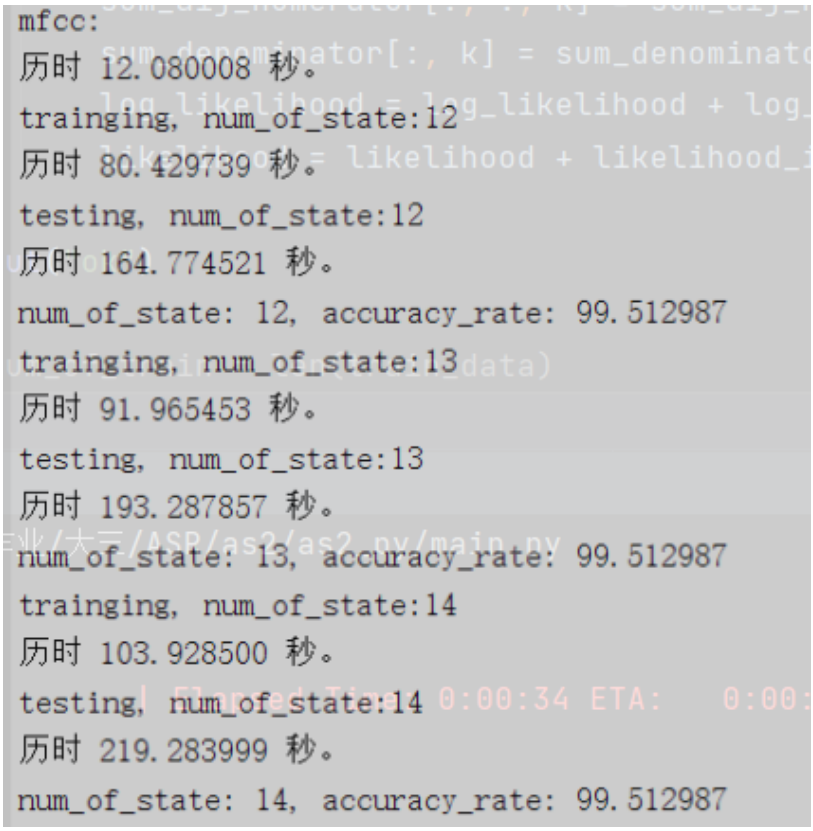
（1）自己实现的mfcc，未加入并行优化

```
采用自己实现的mfcc生成mfcc sample
time cost: 11.959140062332153 s

num_of_state: 12
Training: 100% |##############################################| Elapsed Time: 0:13:01 Time:  0:13:01
Testing:  100% |##############################################| Elapsed Time: 0:17:05 Time:  0:17:05 Acc: 99.43%

num_of_state: 13
Training: 100% |##############################################| Elapsed Time: 0:15:08 Time:  0:15:08
Testing:  100% |##############################################| Elapsed Time: 0:20:11 Time:  0:20:11 Acc: 99.51%

num_of_state: 14
Training: 100% |##############################################| Elapsed Time: 0:17:45 Time:  0:17:45
Testing:  100% |##############################################| Elapsed Time: 0:22:52 Time:  0:22:52 Acc: 99.51%
```

（2）调用库函数的mfcc，加入了并行优化

```
采用mfcc库函数生成mfcc sample
time cost: 10.66321349143982 s

num_of_state: 12
Training: 100% |##############################################| Elapsed Time: 0:04:06 Time:  0:04:06
Testing:  100% |##############################################| Elapsed Time: 0:04:59 Time:  0:04:59 Acc: 99.43%

num_of_state: 13
Training: 100% |##############################################| Elapsed Time: 0:04:48 Time:  0:04:48
Testing:  100% |##############################################| Elapsed Time: 0:05:52 Time:  0:05:52 Acc: 99.51%

num_of_state: 14
Training: 100% |##############################################| Elapsed Time: 0:05:23 Time:  0:05:23
Testing:  100% |##############################################| Elapsed Time: 0:06:35 Time:  0:06:35 Acc: 99.51%

num_of_state: 15
Training: 100% |##############################################| Elapsed Time: 0:10:44 Time:  0:10:44
Testing:  100% |##############################################| Elapsed Time: 0:08:05 Time:  0:08:05 Acc: 99.43%
```

5.2 MATLAB运行截图



5.3 性能分析

5.3.1 时间性能

Python上图是未加入并行时的运行截图，可以发现，和matlab相比，在未加入并行优化前python运行速度是极其慢的。在加入并行优化后python运行性能有所提高，但是相较而言，依旧不及matlab。我认为这是有三方面原因导致的：

（1）matlab在运行时自动并行计算的。相比较与我自己并行实现而言，我只是在两处方便实现并行执行的地方引入了并行运算，但是还有一些地方实际上是可以优化。而matlab可能会尽量将矩阵运算并行执行，这些matlab优化了但是我没有优化的部分可能使得运行时的速度较慢。

（2）matlab更加重视速度，而numpy会兼顾速度和空间。这是我在网上查阅时所显示的一条内容，不过没有更多的资料支持这条原因。但是，从我个人的体验来说，我认为这是合理的。因为在使用numpy进行计算时，我的内存等资源占用是很少的。但是在使用matlab时，其占用的CPU资源较多。

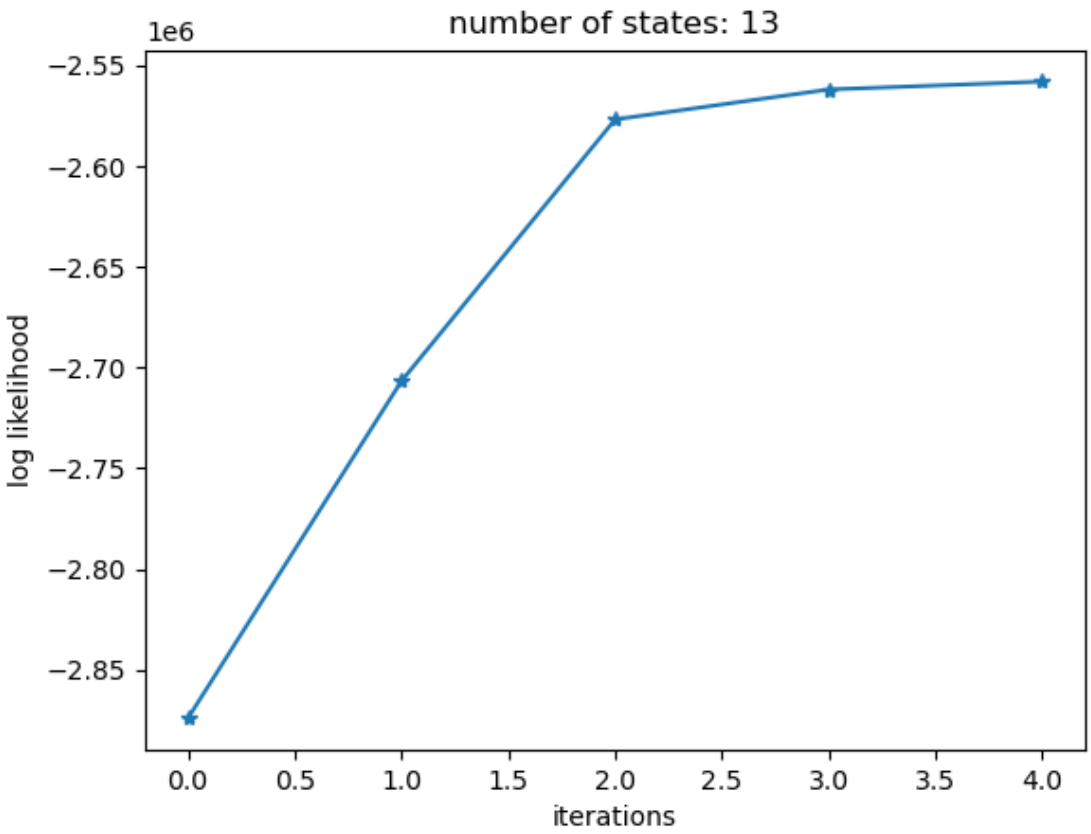
（3）在对矩阵运算的算法优化上，很难有线性代数库可以比得上matlab的，毕竟matlab以矩阵运算闻名天下。

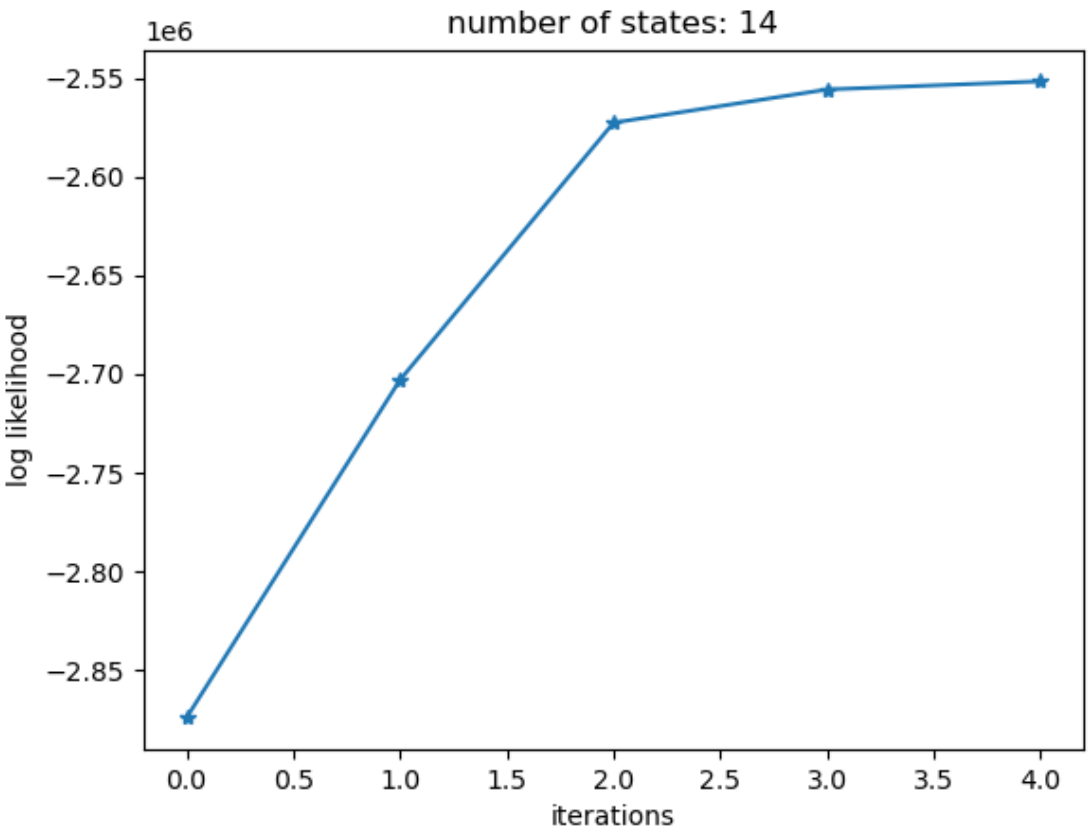
5.3.2 mfcc生成

而由于Matlab和Python代码所生成的mfcc数量级差别很大，导致最大似然估计值差别同样很大，这使得二者虽然绘制曲线的趋势是一样的，但是结果上数量级相差很大。不过尽管matlab和python的mfcc实现不同，最大似然估计也不同，

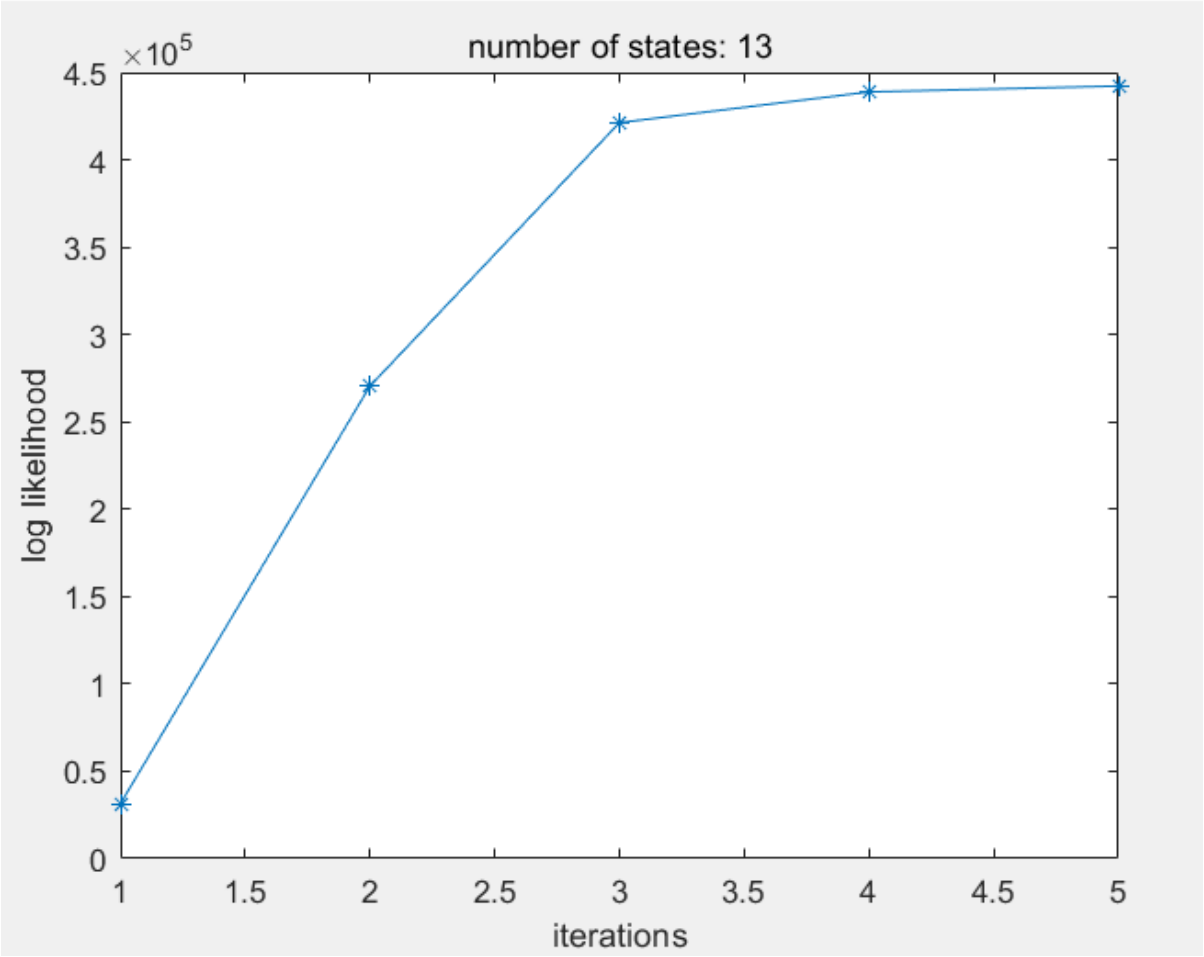
我截取了Python和Matlab中的最大似然训练过程中的两幅图用于对比：

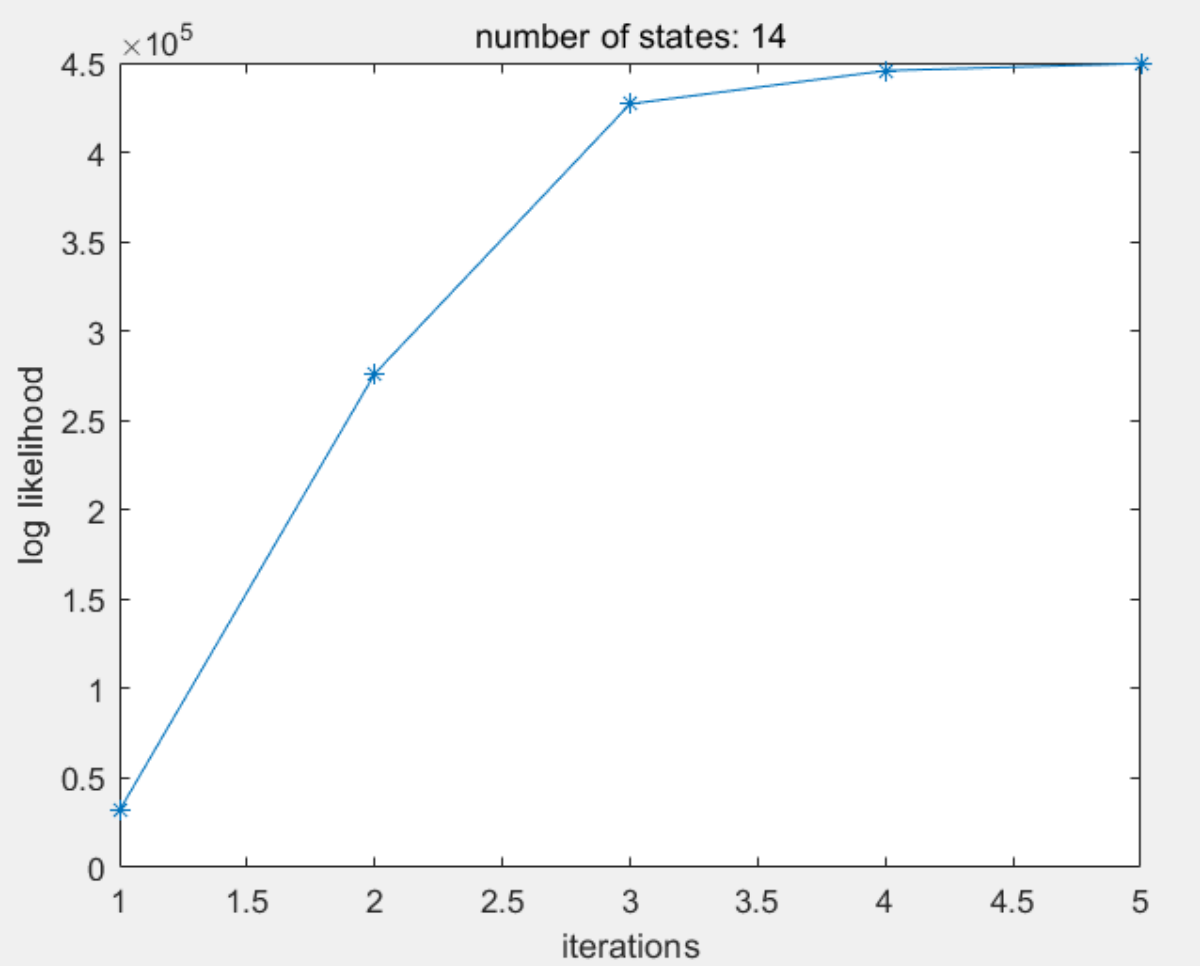
Python：




number of states: 14
1e6
−2.55
−2.60
−2.65
−2.70
−2.75
−2.80
−2.85
log likelihood
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
iterations

Matlab：


number of states: 13
×10^5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
log likelihood
1
1.5
2
2.5
3
3.5
4
4.5
5
iterations



5.3.3 准确率

和matlab相比，python在状态数为12时准确率略逊色与matlab。HMM代码的python实现我参照了matlab的实现，而且在HMM中并没有超参数设置，所以这说明二者之间的差距是在MFCC的实现的，或者在矩阵的数值计算上二者的精度有略微差距。这才导致了准确率的不同。不过，二者差距并不大，只有0.08%左右，这种差距是可以忽视的。

6. 与mfcc库函数进行对比

6.1 时间对比

这里我所选取的库函数是`python_speech_features`中的参数，相比与我自己实现的mfcc来说快了1s。不过，在之前测试中，还存在有mfcc库函数运行16s的情况。所以我认为，这个运行时间与电脑运行时CPU的状态有很大关系。这个1s差距相对来讲并非很大，且从源码对比中发现，我所实现的mfcc和库函数中实现的mfcc的基本流程、算法大致是一样的，除了mel滤波时的参数不完全相同。

因此，可以认为，1s的时间差是可以容忍的，在效率上二者近乎等价。

```
采用mfcc库函数生成mfcc sample
time cost: 10.66321349143982 s
```

```
采用自己实现的mfcc生成mfcc sample
time cost: 11.959140062332153 s
```

6.2 准确率对比

我实现的mfcc和库函数的mfcc的准确率是完全一样的（在保留两位小数的情况下）。

状态数	准确率
12	99.43%
13	99.51%
14	99.51%
15（库函数）	99.43%

这可能是与实现有关，因为二者实现的流程基本相同。在实现自己的mfcc时，我经常纠结各种参数的选取，因为资料上是众说纷纭。不过现在看来是没有必要的，各个参数可能都有道理，但是最终应用在实际上效果差距可能并不会非常显著。如果将准确率向后扩展几位的话二者可能有差别，不过我认为精确到小数点后两位已经足够了。可以认为两个mfcc实现虽然参数选取上不是完全相同，但是效果上却大致一样。

6.3 状态数

从库函数准确率岁状态数的变化来看，并非是状态数越多越好，状态的选取会影响准确率。具体影响需要经过实验，但是这个模型运行时间过长，并不太适合测试。