本次的语音识别作业是训练一组隐马尔科夫模型，最后测试它们识别孤立词的准确率。下面我来介绍一下项目以及它当中各个函数的作用。该项目主要由两个文件组成，HMMModel.py 和 model_main.py 组成。前者是实现了隐马尔科夫链模型以及与它相关的三个算法问题：预测问题，学习参数问题以及解码问题。后者则是将语音识别加入到了这个模型当中，通过一系列手段训练出孤立词的马尔可夫模型，并对它们进行测试实验。

1 model_main.py

首先我们不看隐马尔科夫模型，关注整个语音声学模型的训练流程及其测试。也就是 model_main.py 中的内容。这个文件包括了从提取音频特征值到测试声学模型的一整个过程。主要步骤分为：

1）获得音频文件
2）提取特征值并保存
3）定义模型并将特征值加入到模型中进行训练
4）将训练好的模型用测试文件进行测试，计算语音识别准确率。

获得音频文件：

用到的函数是 generate_wav 函数，它的作用是将 wav 文件读取出来并保存到字典去，这里要注意的是有两个字典，一个是用来存放真正的 wav 文件的 wavdict，另一个则是标签字典，用来对 wav 文件进行分类。因为这里是通过绝对路径来读的，并且在赋值标签字典时用到了正则表达式中字符串分割和截取操作，比较复杂，因此有可能在其他电脑中跑不起来，这时就需要将下图中画红圈的地方改变，可以看到我是根据/,\以及_这三个字符来分割的，目的是得到诸如“1B_endpt.wav”这个文件名的第一个字符，然后将其作为标签。

```
# 生成wav字典并作为返回值返回
def generate_wav(wavpath):
    dictionaryWav = {}
    dictionaryLabel = {}
    for (dirpath, dirnames, filenames) in os.walk(wavpath):
        for filename in filenames:
            if filename.endswith('.wav'):
                WaveFile = os.sep.join([dirpath, filename])
                fileid = WaveFile
                dictionaryWav[fileid] = WaveFile
                label=re.split(r'[/\\_]\s*', WaveFile)[7]
                label=label[0]
                dictionaryLabel[fileid] = label
    return dictionaryWav, dictionaryLabel
```

提取特征值并保存：

这个就不需要多讲了，首先获得 13 维的特征向量，然后将其做一阶差分和二阶差分，最后得到 39 维的 MFCC 特征值。

```python
# 提取MFCC特征值
def getMFCC(file):
    fs, audio = wavfile.read(file)
    mfcc_feat = mfcc(audio, samplerate=fs, numcep=13, winlen=0.025, winstep=0.01, nfilt=26, nfft=2048, lowfreq=0,
                     highfreq=None, preemph=0.97)
    d_mfcc_feat = delta(mfcc_feat, 1)
    d_mfcc_feat2 = delta(mfcc_feat, 2)
    feature_mfcc = np.hstack((mfcc_feat, d_mfcc_feat,d_mfcc_feat2))
    #最终得到一个39维的特征向量
    return feature_mfcc
```

定义模型并将特征值加入到模型中进行训练:

这是整个代码最关键的部分，首先是写一个训练模型 TrainModel 类，如下:

```python
class TrainModel():
    def __init__(self, words=None, n_mix=14, cov_type='full', n_iter=20):
        super(TrainModel, self).__init__()
        print(words)
        self.words = words
        self.category = len(words)
        self.n_mix = n_mix
        self.cov_type = cov_type
        self.n_iter = n_iter
        # models为11个模型的容器
        self.models = []
        for k in range(self.category):
            model = hmm.GaussianHMM(self.n_mix,39,10)
            self.models.append(model)
```

我们在这个新建的类里面建立一个名叫 models 的容器，用来存放要训练的 11 个 HMM 模型，然后将其初始化，从上面我们可以看到，迭代次数为 20，状态数为 14（这个取值 12-15 都可以，因为每一个音素基本上可视作有三个状态，为了应付多音素的孤立词（实际上在我们的测试和训练集里面并没有），就将其定义为 14 个，保证该状态数绝对大于实际状态数）。

接着就是模型的训练函数，如下图所示:

```python
# 模型训练
def train(self, wavdict=None, labeldict=None):
    count=np.zeros(11)
    for k in range(11):
        for x in wavdict:
            if labeldict[x] == self.words[k]:
                mfcc_featSingle = getMFCC(wavdict[x])
                count[k]=count[k]+mfcc_featSingle.shape[0]
        print(count)
    print("样本已分离")
    samples_1 = np.zeros((int(count[0]), 39))
    samples_2= np.zeros((int(count[1]), 39))
    samples_3 = np.zeros((int(count[2]), 39))
    samples_4 = np.zeros((int(count[3]), 39))
    samples_5 = np.zeros((int(count[4]), 39))
    samples_6 = np.zeros((int(count[5]), 39))
    samples_7 = np.zeros((int(count[6]), 39))
    samples_8 = np.zeros((int(count[7]), 39))
    samples_9 = np.zeros((int(count[8]), 39))
    samples_O = np.zeros((int(count[9]), 39))
    samples_Z = np.zeros((int(count[10]), 39))
    for k in range(11):
        for x in wavdict:
            if labeldict[x] == self.words[k]:
                if (k == 0):
```

这里我用得到的方法其实是比较麻烦的，因为每一种音频的总帧数都是不一样的，就以训练样本为前 10 个文件夹中的 220 个音频文件为例，比如 one 这个单词，所有的训练样本构成的 mfcc 特征值差不多是一个 39*1061 一个矩阵，而 two 这个单词的特征值就比它少，只有 866 个即 39*866 矩阵，如下图：

```
        print(count)
    print("样本已分离")
    samples_1 = np.zeros((int(count[0]), 39))
    samples_2= np.zeros((int(count[1]), 39))
    samples_3 = np.zeros((int(count[2]), 39))
    samples_4 = np.zeros((int(count[3]), 39))
    samples_5 = np.zeros((int(count[4]), 39))
```

TrainModel

model_main

```
[1061.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
[1061.  866.    0.    0.    0.    0.    0.    0.    0.    0.]
[1061.  866.  813.    0.    0.    0.    0.    0.    0.    0.]
[1061.  866.  813.  987.    0.    0.    0.    0.    0.    0.]
[1061.  866.  813.  987. 1182.    0.    0.    0.    0.    0.]
[1061.  866.  813.  987. 1182. 1297.    0.    0.    0.    0.]
[1061.  866.  813.  987. 1182. 1297. 1114.    0.    0.    0.]
[1061.  866.  813.  987. 1182. 1297. 1114.  834.    0.    0.]
[1061.  866.  813.  987. 1182. 1297. 1114.  834. 1132.    0.]
[1061.  866.  813.  987. 1182. 1297. 1114.  834. 1132.  925.]
[1061.  866.  813.  987. 1182. 1297. 1114.  834. 1132.  925. 1213.]
样本已分离
```

这就导致我不能用一个 3 维矩阵去将它们 11 种样本存起来，否则除了最多的那个单词的特征值被利用以外，其他的都哦加入了“杂质”，即 0 数字，这对于训练模型是不利的，因此我再没有想到更好的办法的情况下只能新建了 11 个变量来存它们的特征样本值，然后再不断遍历它们，这里的算法效率不能说低，毕竟是手写的 if 语句和 for 语句，但是代码可读性会变得比较差一些。

将训练好的模型用测试文件进行测试，计算语音识别准确率。

这里就是利用维特比算法来计算最大概率，选出拥有最大概率的模型，然后将标签作为返回值返回。

```python
    # 对测试wav字典使用维特比算法，然后得到最大概率模型所对应的标签
    def viterbi(self, filepath):
        result = []
        for k in range(self.category):
            model = self.models[k]
            data1 = []
            data2 = []
            mfcc_feat = getMFCC(filepath)
            destination = model.score(mfcc_feat)
            data1.append(destination)
            result.append(data1)
            #print(destination)

        # 通过维特比算法得到最大概率模型所对应的标签
        result = np.vstack(result).argmax(axis=0)
        result = [self.words[label] for label in result]
        #print('得到的标签为：\n',result)
        return result
```

HMMModel.py

然后我们来看隐马尔可夫链模型，我主要参照了李航的《统计方法学习》一书中的内容，将前向概率、后向概率、状态转移概率、观测概率（也叫发射概率）、EM 迭代计算等计算公式写成各个函数，最后实现维特比算法也就是解码算法。

在这里关于模型各种概率的解释我在 README.md 里说过，在这就不再赘述。下面分别进行介绍函数的作用：

初始化 GMMHMM 模型，之所以用到高斯模型，是因为我们的音频特征值属于连续的向量，而它又作为 HMM 模型的观测值 O。

```python
#定义连续分布即高斯分布下的HMM模型，这是因为我们的音频特征值属于连续的向量，而它又作为HMM模型的观测值。
class GMMHMM:
    def __init__(self, state_num=1, x_len=1, itertimes_num=20):
        self.state_num = state_num
        self.itertimes_num = itertimes_num
        self.x_len = x_len
        # 初始化观测概率均值
        self.means_B = np.zeros((state_num, x_len))
        # 初始化观测概率协方差
        self.covars_B = np.zeros((state_num, x_len, x_len))
        # 初始化为均值为0，方差为1
        for i in range(state_num): self.covars_B[i] = np.eye(x_len)
```

利用前后向概率的统计量（一系列的求和运算）得到观测概率

```
    # 求x在状态k下的观测概率
    def prob_B(self, x):
        prob = np.zeros((self.state_num))
        for i in range(self.state_num):
            prob[i]=gauss2D(x, self.means_B[i], self.covars_B[i])
        return prob
```

在训练过程中需要更新观测概率，因为这是一个不停迭代的过程

```
    # 求x在状态k下的观测概率
    def prob_B(self, x):
        prob = np.zeros((self.state_num))
        for i in range(self.state_num):
            prob[i]=gauss2D(x, self.means_B[i], self.covars_B[i])
        return prob
```

然后便是进行无监督学习，通过迭代学习，不断将观测概率、状态转移概率收敛到一个正确的局部最优解（其实是观测矩阵和状态转移概率才对），其中迭代用到的是 EM 算法，即先求极大似然函数，这是 E 步骤，然后进行收敛参数，这是 M 步骤。

```
# 对观测序列的训练
def fit(self, X, Z_seq=np.array([])):
    # 输入X观测序列
    # 输入Z为未知的状态序列，待求
    self.trained = True
    X_length = len(X)
    self._init(X)

    # 对状态序列进行初始化
    if Z_seq.any():
        Z = np.zeros((X_length, self.state_num))
        for i in range(X_length):
            Z[i][int(Z_seq[i])] = 1
    else:
        Z = np.ones((X_length, self.state_num))
    # EM步骤迭代
    for e in range(self.itertimes_num):
        print(e, " iter")
        # 求期望即最大似然
        # 向前向后传递因子
        alpha, c = self.forward(X, Z)  # P(x,z)
        beta = self.backward(X, Z, c)  # P(x|z)

        post_state = alpha * beta
        post_adj_state = np.zeros((self.state_num, self.state_num))  # 相邻状态的联合后验概率
```

运用递归的方法求前后向概率值，根据统计学习方法中的介绍，这里的递归使得算法复杂度从原来的$O(TN^T)$降到了$O(TN^2)$，其中 T 是观测时间，从 t=1 取到 t=T，N 是可能的状态数。

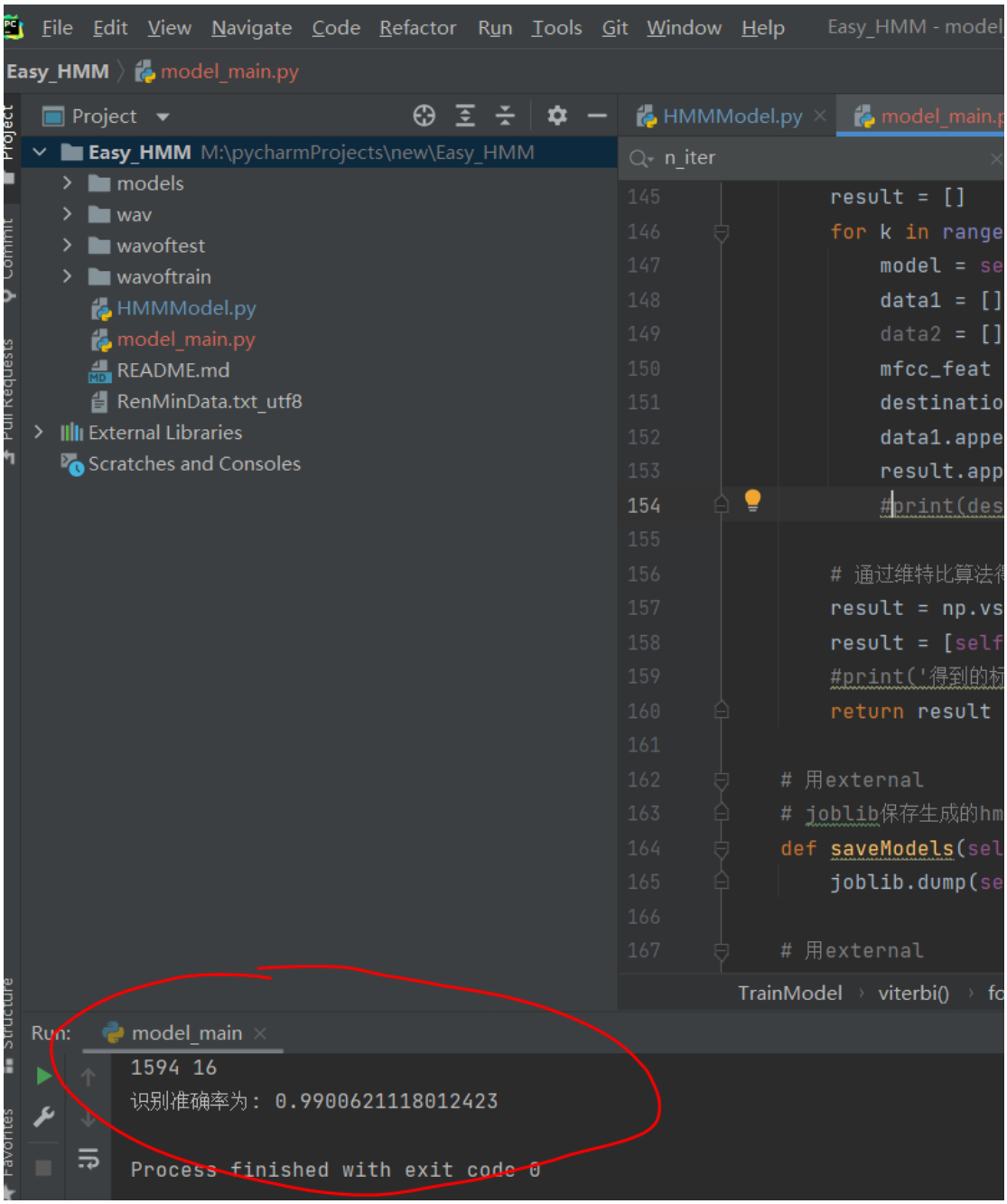
```python
# 利用HMM前向算法求前向概率
def forward(self, X, Z):
    X_length = len(X)
    alpha = np.zeros((X_length, self.state_num))  # P(x,z)
    alpha[0] = self.prob_B(X[0]) * self.start_prob * Z[0]  # 初始值
    # 归一化因子
    c = np.zeros(X_length)
    c[0] = np.sum(alpha[0])
    alpha[0] = alpha[0] / c[0]
    # 递归传递
    for i in range(X_length):
        if i == 0: continue
        alpha[i] = self.prob_B(X[i]) * np.dot(alpha[i - 1], self.transmat_prob) * Z[i]
        c[i] = np.sum(alpha[i])
        if c[i] == 0: continue
        alpha[i] = alpha[i] / c[i]

    return alpha, c
```

```python
#利用后向算法求后向概率
def backward(self, X, Z, c):
    X_length = len(X)
    beta = np.zeros((X_length, self.state_num))  # P(x|z)
    beta[X_length - 1] = np.ones((self.state_num))
    # 递归传递
    for i in reversed(range(X_length)):
        if i == X_length - 1: continue
        beta[i] = np.dot(beta[i + 1] * self.prob_B(X[i + 1]), self.transmat_prob.T) * Z[i]
        if c[i + 1] == 0: continue
        beta[i] = beta[i] / c[i + 1]

    return beta
```

最后是用维特比算法进行状态序列的预测，然后得到匹配的最大概率

```python
def viterbi(self, X, istrain=True):
    if self.trained == False or istrain == False:  # 需要根据该序列重新训练
        self.fit(X)

    X_length = len(X)  # 序列长度
    state = np.zeros(X_length)  # 隐藏状态

    pre_state = np.zeros((X_length, self.state_num))  # 保存转换到当前隐藏状态的最可能的前一状态
    max_pro_state = np.zeros((X_length, self.state_num))  # 保存传递到序列某位置当前状态的最大概率

    _,c=self.forward(X, np.ones((X_length, self.state_num)))
    max_pro_state[0] = self.prob_B(X[0]) * self.start_prob * (1 / c[0])# 初始概率

    # 前向求和的概率
    for i in range(X_length):
        if i == 0: continue
        for k in range(self.state_num):
            prob_state = self.prob_B(X[i])[k] * self.transmat_prob[:, k] * max_pro_state[i - 1]
            max_pro_state[i][k] = np.max(prob_state)* (1/c[i])
            pre_state[i][k] = np.argmax(prob_state)

    # 后向求和的概率
    state[X_length - 1] = np.argmax(max_pro_state[X_length - 1,:])
    for i in reversed(range(X_length)):
        if i == X_length - 1: continue
        state[i] = pre_state[i + 1][int(state[i + 1])]
```

下面这是训练了 10 个文件中 220 个音频的模型，最后得到的准确率为 99.006%



下面这是训练了 10 个文件中 220 个音频的模型，最后得到的准确率为 99.316%

可见，训练的样本越多，效果就越好。

在用自己的 mfcc 函数提取特征值的时候，效果非常的差，大概只有不到一半的准确率，但是自己也没能找出来原因所在。