Assignment 2——Hidden Markov Models

- 学号：1952335
- 姓名：代玉琢

一、项目描述

本次项目是根据给定的HMM的 *MATLAB* 文件，完成一个使用 *python* 语言的HMM模型。基本功能包括根据训练样本文件生成11个HMM模型，并使用测试样本测试其准确率。

文件架构

├─.idea
├─list_by_librosa：使用librosa的样本路径
├─list_by_myself：使用自己的mfcc的样本路径
├─list_by_psf：使用python_speech_features的样本路径
├─l_mfcc：使用librosa生成的特征
├─mfcc：使用python_speech_features生成的特征
├─my_mfcc：使用自己写的mfcc生成的特征
├─test_librosa_1：使用librosa，迭代一次的结果
├─test_psf_5：使用python_speech_features，迭代五次的结果
├─venv
└─__pycache__

二、函数过程描述

1.生成mfcc特征：

1) *generate_mfcc_samples()*

对wav文件夹下的所有样本生成mfcc特征。
使用 ***python_speech_features*** 的 mfcc 接口生成 mfcc 特征，保存到 mfcc 文件夹中。

2) *generate_mfcc_samples_by_myself()*

对wav文件夹下的所有样本生成 mfcc 特征。
使用自己写的文件生成 mfcc 特征，保存到 my_mfcc 文件夹中。

3) *generate_mfcc_samples_by_librosa()*

对wav文件夹下的所有样本生成 mfcc 特征。
使用 ***librosa***包中的 mfcc 接口生成 mfcc 特征，保存到 l_mfcc 文件夹中。

2.生成训练数据：

1)*generate_training_list_mat()*

选择一定数量的样本作为训练数据，保存在 *training_list* 中，文件格式为csv。在这里，选择了**440**个样本作为训练样本。

3.生成测试数据：

1) *generate_testing_list_mat()*

选择一定数量的样本作为训练数据，保存在 *testing_list* 中,文件格式为csv。在这里，选择**1232**个样本作为测试样本。

4.一些初始化工作

1. 确定维度：**DIM=39**
2. 模型的数量：**num_of_model = 11**（对于*1~9*，*zero*和*O*）
3. 训练设置的状态：**12~15**
4. 准确率：**accuracy_rate**

然后进入HMM模型的训练，针对确定的状态数，每次训练出11个模型，并测试其准确率accuracy_rate。

5.模型的训练

EM_HMM_training

这个函数的主要功能就是对训练样本使用EM算法训练模型，具体功能由其中的几个核心函数实现。最终得到是一个HMM的三个参数：均值矩阵、方差矩阵和状态转移矩阵。

实际上，整个HMM模型的训练的目的是为了求两个重要参数：a_{ij} 和 $b_j(x_t)$。其中，*Aij* 分别是11个模型的状态转移矩阵 a_{ij}，而 $b_j(x_t)$ 可以将 mean 和 var 带入高斯公式计算求得。

1)Step 1：模型的初始化

初始化HMM的三个参数：均值矩阵、方差矩阵、状态转移概率矩阵。
mean： num_of_model X DIM X num_of_state
var： num_of_model X DIM X num_of_state
Aij： num_of_model X num_of_state+2 X num_of_state+2（增加了一个start状态和一个end状态）

EMM_initialization & calculate_initial_EM_HMM_items

这两个函数都是用于初始化HMM模型的，前者调用了后者完成HMM模型的初始化。

EMM_initialization(mean, var, Aij, training_file_list_name, DIM, num_of_state, num_of_model)：
这个函数对模型进行初始化，初始化HMM的三个参数（均值矩阵、方差矩阵、状态转移矩阵），根据训练样本，计算所有训练样本的之和。因为我们得到的MFCC特征是 维数 × 帧数 。
在 *EMM_initialization* 函数里，主要是计算了特征的总和(*sum_of_features*)、特征的平方的总和(*sum_of_features_square*)和特征总的帧数(*num_of_feature*)。

```
temp = np.sum(features, axis=1)
sum_of_features = np.add(sum_of_features, temp)
temp = np.sum(np.square(features), axis=1)
sum_of_features_square = np.add(sum_of_features_square, temp)
num_of_feature = num_of_feature + features.shape[1]  # 特征的总的帧数
```

然后在 *calculate_initial_EM_HMM_items* 函数里利用这些值计算出均值矩阵、方差矩阵和状态转移矩阵。

calculate_initial_EM_HMM_items(mean, var, Aij, num_of_state, num_of_model, sum_of_features,sum_of_features_square, num_of_feature)

通过前述可知，这个函数的作用是利用计算好的值来初始化HMM的三个参数。

均值矩阵的初始化是将所有特征按列相加(*sum_of_features*)，再除以特征的总帧数(*num_of_feature*)。

方差矩阵的初始化则是将所有特征按列平方再相加(*sum_of_features_square*)，再除以特征的总帧数数(*num_of_feature*)。

状态转移矩阵的初始化是默认从 *i* 到 *i+1* 的概率为***0.4***，从 *i* 到 *i* 的概率是***0.6***，并且从状态 *0* 到状态 *1* 的概率是***1***。

```
for i in range(num_of_model):
    for j in range(num_of_state):
        # 初始化平均值和方差
        mean[i, :, j] = sum_of_features / num_of_feature
        var[i, :, j] = sum_of_features_square / num_of_feature -
np.square(mean[i, :, j])
    for k in range(1, num_of_state + 1):
        # 初始化状态转移矩阵
        Aij[i, k, k + 1] = 0.4  # 从k到k+1的概率初始化为0.4
        Aij[i, k, k] = 1 - Aij[i][k][k + 1]  # 从k到k的概率初始化为1-0.4=0.6
    Aij[i, 0, 1] = 1  # 初始化时，初始概率为1
```

在对HMM进行初始化后，设置好**迭代次数**，就正式开始模型的训练了。

2) Step 2：模型的训练

在迭代训练中，我们需要计算的**几个参数**是：均值矩阵的分子、方差矩阵的分子、状态转移矩阵的分子以及它们共同的分母。均值矩阵和方差矩阵计算的主要目的是用于计算发射概率，即$b_j(x_t)$。

由理论学习可知，a_{ij} 和 $b_j(x_t)$ 的公式如下：

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t{}^{(i,j)}}{\sum_{t=1}^{T-1} \gamma_t^j}$$

$$\hat{b}_j(x) = \frac{\sum_{t=1}^{T} \gamma_t^j \cdot 1(x = x_t)}{\sum_{t=1}^{T} \gamma_t^j}$$

上面需要计算的参数在 *EM_HMM_FR* 函数中计算。即利用EM算法进行模型的训练。

EM_HMM_FR(mean, var, Aij, obs)

首先，因为直接计算得到的数会很小，为了提高准确率，在计算的过程中实际上是对整体取对数，即将乘法变成了加法。

由上述公式可知，在训练的过程中，我们需要计算的实际上是利用初始化的 a_{ij} 和$b_j(x_t)$ （*Aij*，*mean*，*var*）进行E-step，即计算α、β、γ、ξ，然后在M-step再重新计算 a_{ij} 和 $b_j(x_t)$。

在训练的过程中，主要依赖于以下几个函数：

log_sum_alpha(log_alpha_t, aij_j)

这个函数在计算 α 的迭代中很重要，用于计算 $\sum_{i=1}^{N} \alpha_{t-1}^j a_{ij}$。

可以看作：*log_sum_alpha(log_alpha_t, aij_j)* = $\log \sum_{i=1}^{N} \alpha_{t-1}^j a_{ij}$

log_sum_beta(aij_i, mean, var, obs, beta_t)

这个函数在计算 β 的迭代中很重要，用于计算 $\sum_{j=1}^{N} a_{ij} b_j(x_t) \beta_t^j$。

可以看作：*log_sum_beta(aij_i, mean, var, obs, beta_t)* = $\sum_{j=1}^{N} a_{ij} b_j(x_t) \beta_t^j$。

logGaussian(mean_i, var_i, o_t)

这个函数是利用高斯公式计算发射概率（观察概率）：$b_j(x_t)$。

可以理解为：*logGaussian(mean_i, var_i, o_t)* = $\log b_i(x_t)$。

***E-M*算法：**

a.计算 $\log \alpha$

计算步骤：

step 1：初始化

$$\alpha_1^j = p(x_1 \mid q_1 = j)p(q_1 = j)$$

step 2：迭代

$$\alpha_t^j = b_j(x_t)\sum_{i=1}^{N}(\alpha_(t-1)^i \cdot a_i j), 1 \le j \le N, 2 \le t \le T$$

step 3：终止

$$\sum_{j=1}^{N}\alpha_T^j$$

代码如下：

```
for i in range(N):  # 初始化第一列
    log_alpha[i, 0] = np.log(Aij[0, i]) + logGaussian(mean[:, i], var[:, i],
obs[:, 0])
    # mean[:][i]，var[:][i]：状态i下的平均值和方差

for t in range(1, T):  # t=1~T-1
    for j in range(1, N - 1):  # j=1~N-2
        # log_sum_alpha(log_alpha(2:N-1,t-1),aij(2:N-1,j))
        log_alpha[j, t] = log_sum_alpha(log_alpha[1:N - 1, t - 1], Aij[1:N - 1,
j]) + logGaussian(mean[:, j],
```

b.计算 $\log \beta$

计算步骤：

step 1：初始化

$$\beta_1^j = A_{iT}$$

step 2：迭代

$$\beta_{t-1}^i = \sum_{j=1}^{N} a_{ij}b_j(x_t)\beta_t^j$$

step 3：终止

$$\sum_{i=1}^{N} a_{0i}b_i(x_1)\beta_1^i$$

代码如下：

```
log_beta[:, T - 1] = np.log(Aij[:, - 1])  # 初始化T时刻的β
for t in range(T - 2, -1, -1):  # t=T-2~0
    for i in range(1, N - 1):  # i=1~N-2
        log_beta[i, t] = log_sum_beta(Aij[i, 1:N - 1], mean[:, 1:N - 1], var[:,
1:N - 1], obs[:, t + 1],log_beta[1:N - 1, t + 1])
log_beta[N - 1, 0] = log_sum_beta(Aij[0, 1:N - 1], mean[:, 1:N - 1], var[:, 1:N
- 1], obs[:, 0],
                                  log_beta[1:N - 1, 0])
```

c.计算 $\log\gamma$

$$\gamma_t^j = \frac{\alpha_t^j \beta_t^j}{\sum_{j=1}^N \alpha_t^j \beta_t^j}$$

代码如下：

```
log_gamma = np.array(np.ones((N, T)) * -np.inf)
for t in range(T):
    for i in range(1, N - 1):
        log_gamma[i, t] = log_alpha[i, t] + log_beta[i, t] - log_alpha[N - 1, T]
gamma = np.exp(log_gamma)
```

d.计算 $\log\xi$

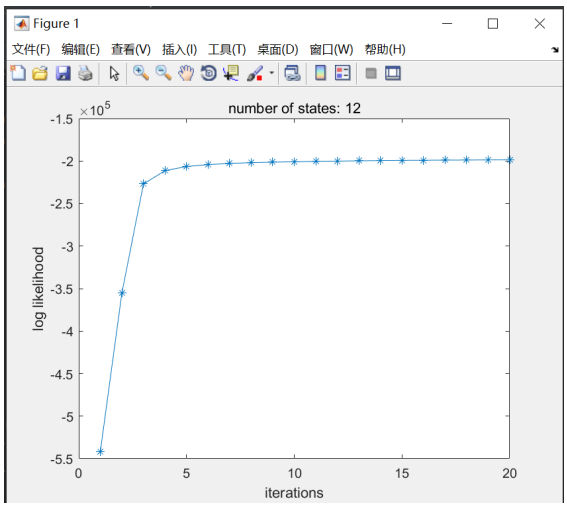
$$\xi_t^{(i,j)} = \frac{\alpha_t^i a_{ij} b_j(x_{t+1}) \beta_{t+1}^j}{\sum_{j=1}^N \alpha_t^j \beta_t^j}$$

代码如下：

```
log_Xi = np.array(np.ones((T, N, N)) * -np.inf)
for t in range(T - 1):
    for j in range(1, N - 1):
        for i in range(1, N - 1):
            log_Xi[t, i, j] = log_alpha[i, t] + np.log(Aij[i, j]) +
logGaussian(mean[:, j], var[:, j], obs[:, t + 1]) + log_beta[j, t + 1] -
log_alpha[N - 1, T]
# 计算T时刻的ξ，即t=T-1
for i in range(N):
    log_Xi[T - 1, i, N - 1] = log_alpha[i, T - 1] + np.log(Aij[i, N - 1]) -
log_alpha[N - 1, T]
```

在这里，就完成了E-step。
然后就进入了M-step，即使用E-step计算的 α、β、γ、ξ 再次计算均值矩阵、方差矩阵、状态转移矩阵的分子以及它们的公共分母。
这就是一个完整的EM算法。

总结来说，在***EM_HMM_FR***里完成的功能就是使用一次EM算法。其中，*E-step* 就是使用初始化（或已有的）HMM参数来计算 α、β、γ、ξ，这个过程主要是使用了一个迭代的计算方法。然后，在*M-step* 就是利用在 *E-step* 计算得到的α、β、γ、ξ 来再次计算出HMM的参数。多次使用EM算法，即多次迭代，最后可以得到一个收敛的模型。通过对 *matlab* 源码的测试，可以发现在迭代次数 >5 时，模型逐渐趋于收敛。



通过计时，了解到一次迭代的时间在200~500s不等，为了减少训练时间，所以我在测试阶段选择的迭代次数是5。
最后，通过多次迭代，得到了HMM模型的三个参数：均值、方差、状态转移概率矩阵。

6.测试模型

HMM_testing

HMM_testing(mean, var, Aij, testing_list_name)
对于训练出的模型，还需要对该模型进行测试。
这个函数的主要功能是输入HMM的三个参数，最终输出一个 *accuracy_rate* ，即模型的准确率。其中的核心算法就是 *Viterbi* 算法，即计算哪个模型的概率(f_opt)最大，并且可以找到一个最佳状态链(chain_op)。
在载入测试样本后，就开始对每一个测试样本进行测试。

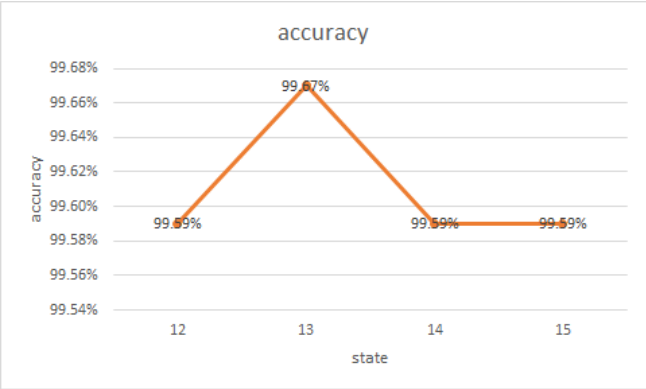
viterbi_dist_FR(mean, var, aij, obs)
这个函数就是测试模型的核心函数。可以把它理解为一个解码(*decoding*)的过程。根据输入的HMM模型和MFCC特征，计算该语音是该模型的最大概率，及其最优的状态链。
通过 *Viterbi* 算法得到一个特征属于各个模型的概率，选择概率最大的模型。如果和测试样本的发音一致，则说明正确，否则错误。最后通过 **样本总数-错误次数/样本总数**，得到*accuracy_rate*。

7.对比实验

1）不同状态数对比

在测试中，模型的状态数范围为：12~15。每一次迭代训练的时间在200~400s不等，测试时间在40分钟左右。最终得到如下的结果：

state	12	13	14	15
accuracy	99.59%	99.67%	99.59%	99.59%

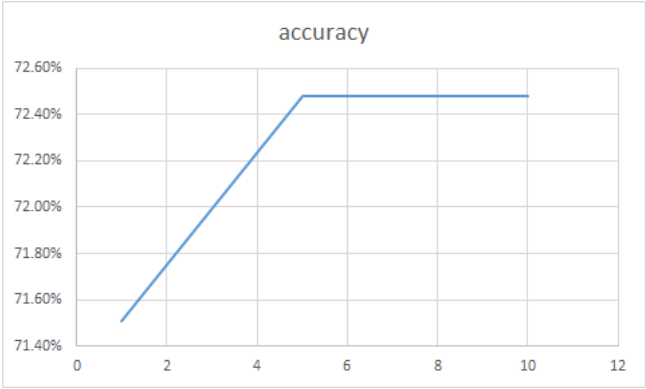


可以发现，在状态数选择为13时，模型的准确率是最高的，所以选择13个状态训练模型更好。

2）不同迭代次数对比

在测试阶段，为了更快地训练模型，我在测试时选择的迭代次数都是1。通过 matlab 的测试，了解到迭代次数在大于5之后，就趋于收敛了，所以在这里我对比一下不同迭代次数的模型效果。因为使用Librosa训练的速度较快，所以用它作为实验样本，并且状态数为12。最终结果如下：

itertation	1	5	10
accuracy	72%	72.48%	72.48%

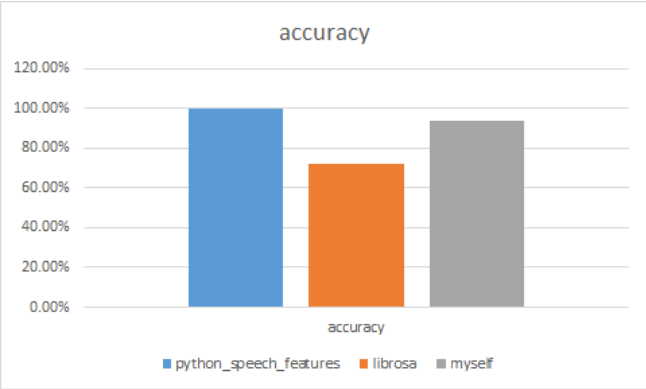


可以看到，在迭代次数较少时，模型还未收敛，准确率较低。但是当迭代次数达到一定数量时，模型已经收敛，再多次的迭代意义不大。

3）不同的MFCC对比

通过查阅资料，我了解到python可以通过两种方式调用mfcc接口，一种是使用librosa包里的mfcc函数，另一种是调用python_speech_features里的mfcc接口。所以我最终选择了这两种mfcc接口以及自己写的mfcc特征code来对比最终生成的HMM模型效果。为了减少测试时间，只考虑状态数为12的情况，每个mfcc接口都迭代训练了5次。最终结果如下：

MFCC	python_speech_features	librosa	myself
accuracy	99.59%	72.48%	93.75%



通过上表和上图可以知道，python_speech_features得到的mfcc特征训练出的模型准确率最高，我自己写的mfcc次之，而librosa得到的mfcc特征效果最差。
实际上，我在实验时，发现对于同意一个样本，librosa得到的mfcc特征的帧数是最少的，所以它可能是在特征提取中由于帧移或帧长的问题造成了一些特征的丢失。
而我写的mfcc特征与python_speech_features不同之处在于差分这一步，我自己写的是用的一种简单差分，而p_s_f中我调用了其中的delta函数做差分，在这个过程中可能导致了一些特征的丢失，从而造成了准确率的下降。

8.问题总结

1）翻译代码踩的坑

MATLAB 和 *python* 都是我才接触不久的两门语言，所以这两门语言对我来说都比较生疏，相较于 *python*，我对于 *MATLAB* 更加生疏。

在整个代码翻译的过程中，需要特别注意的几点如下：

a.索引和切片问题

在 *MATLAB*中，矩阵、数组的索引都是从1开始的。而在 *python* 中，这些索引都是从 0 开始的。
在 *MATLAB* 中，切片操作是一个闭区间，即包含索引起始项和结束项。但是，在 *python* 中，切片操作是一个半开半闭的过程，即包含索引起始项，但是不包含索引结束项。

b.数组问题

一维数组、二维数组，在 *MATLAB* 和 *python* 中可以看作是等同的。
但是三维数组在两者中是不同的。如 *[i, j, k]*，在 *MATLAB* 中，是第 k 页第 i 行第 j 列；但是，在 *python* 中则表示第 i 页第 j 行第 k 列。
这两个问题就是我在翻译过程中感受最深的问题，在出现 bug 时，很大概率就是数组或者索引的问题。

其中的一个问题：

完成训练部分后，进入测试阶段。最后得到的准确率竟然是 *0%*，让我很吃惊。
在对比代码后，我发现问题出现在一个嵌套循环里。这个问题就是上面提到的两个问题之一，索引的问题。
在 *MATLAB* 里：

```
for t=2:t_len
    dt = timing(t)-timing(t-1);
    for j=2:m_len-1 %(2->14)
        f_max = -Inf;
        i_max = -1;
        f = -Inf;
        for i=2:j
            if(fjt(i,t-1) > -Inf)
                f = fjt(i,t-1) + log(aij(i,j,dt)) + logGaussian(mean(:,j),var(:,j),obs(:,t));
            end
            if f > f_max % finding the f max
                f_max = f;
                i_max = i; % index
            end
        end
        if i_max ~= -1
            s_chain{j,t} = [s_chain{i_max,t-1} j];
            fjt(j,t) = f_max;
        end
    end
end
```

在第三个 for 循环里：

```
for i=2:j
    ....
```

这里的 i 是从 2 到 j 的，我直接把这条语句翻译成：

```
for i in range(1,j):
    ...
```

最后得到的 *Viterbi* 矩阵是这样的：

这样使得 *Viterbi* 最后输出的概率是 *-inf* ，所以也无法预测相应的模型，自然是预测错误的，造成了计算 accuracy 时的错误。

所以正确的翻译应该是：

```
for i in range(1,j+1):
    ....
```

```
for t in range(1, t_len):
    dt = timing[t] - timing[t - 1]
    for j in range(1, m_len - 1):
        f_max = -np.inf
        i_max = -1
        f = -np.inf
        for i in range(1, j + 1):
            if f_jt[i, t - 1] > -np.inf:
                f = f_jt[i, t - 1] + np.log(aij[i, j]) + logGaussian(mean[:, j], var[:, j], obs[:, t])
            if f > f_max:  # 找到最大的f，为了计算Viterbi矩阵的下一列
                f_max = f
                i_max = i  # 索引
        if i_max != -1:
            s_chain[j, t] = np.append(s_chain[i_max, t - 1], j)
            f_jt[j, t] = f_max
```

最后得到的 *Viterbi* 矩阵如下图所示：

解决后，通过测试，最后得到的 *accuracy_rate *不再是0%了。

2) MFCC问题：

在选择 mfcc 接口时，我对上次作业完成的 mfcc 特征提取做了一点修改，然后进行测试，但是训练出来的模型准确率很低，只有百分之三十几。所以我怀疑自己的写的 mfcc 特征存在一些问题。

采样率的问题

因为当时的代码主要针对的是一个固定的音频，所以为了方便，我直接将采样率给定为：44100Hz。但是这次的训练样本的采样率是8000Hz。以及DFT和 mel 滤波器的一些调整。

结果

状态数为12的准确率: 93.75 %

3) 总结

在本次作业中，我首先花了很多时间来理解HMM模型以及它的三个重要问题，然后花费了其他时间来翻译 *MATLAB* 代码和 debug。在其中，遇到了很多问题，也存在一次完整的训练、测试的时间过长等问题。虽然过程十分痛苦，但是最终的结果是令人感到快乐的。