# 案例三 语音识别

## 【1】案例背景

本题由 **顾翔 921127940122**负责

语音识别是指通过专门的声学模型和语言模型等技术，把声音等音频信号转化为与之对应的文本的过程，主要包括声学的特征，提取模式，匹配模型训练等流程，近些年来语音识别技术得到广泛的应用，如汽车导航语音输入法等，本案例使用的数据及包含7个英语单词的音频文件数据，一共有105个音频文件，请根据提供的音频文件使用隐马尔可夫模型识别出音频中听读的单词

## 【2】方法陈述

首先使用pandas对数据进行处理，接着，使用主成分分析算法（PCA）对数据进行降维，最后将数据展示出来，分析结果。

运行环境，python3安装有pandas、scikit-learn、numpy库，运行软件：Anaconda、jupyter notebook

注：本题使用到的算法（特征处理STFT，训练GMM-HMM），过于复杂，因此代码参考较多

## 【3】实验代码

### 步骤一、数据处理

1. 该次作业数据处理极为复杂，首先,需要从文件汇总.wav数据集，根据语音样本转化为矩阵

2. 接着对数据进行短时傅里叶变换STFT，首先语音将会被分成几个小块，然后对每一小块进行傅里叶变换FFT，得到一个二维的FFT“图片”，这个“图片”即我们通常所说的频谱。选择不同的FFT的大小会得到不同频率分辨率下的结果，而重叠这些窗口可以让我们以增加数据大小为代价来控制时间分辨率。

3. 将STFT后的数据使用滑窗提取峰值，最终得到的是一个 105×6×216 的三维数组，最终每一个语音波形都得到216个重叠帧，每一帧用最大的6个峰值对应的频率来描述。

fpaths = [] *# 文件路径*

labels = [] *# 每一个语音文件所包含的单词内容*

spoken = [] *# 所有语音文件所包含的单词内容列表*

import **os**

for f in **os**.**listdir**('./audio')[0:]:

    for w in **os**.**listdir**('./audio/' + f):

        fpaths.**append**('./audio/' + f + '/' + w)

        labels.**append**(f)

        if f not in spoken:

            spoken.**append**(f)

**print**('Words spoken:', spoken)

from **scipy**.**io** import **wavfile**

import **numpy** as **np**

data = **np**.**zeros**((**len**(fpaths), 32000)) *# 每一行为一个语音样本，每一列为样本的一个特征*

maxsize = -1 *# 最大的列数，每一个音频文件的时长不同，所以在循环中需要不断更改*

for n,file in **enumerate**(fpaths):

    \_, d = **wavfile**.**read**(file)

    data[n, :d.shape[0]] = d

    if d.shape[0] > maxsize:

        maxsize = d.shape[0]

data = data[:, :maxsize]

*#Each sample file is one row in data, and has one entry in labels*

**print**('Number of files total:', data.shape[0])

all\_labels = **np**.**zeros**(data.shape[0])

for n, l in **enumerate**(**set**(labels)):

    all\_labels[**np**.**array**([i for i, \_ in **enumerate**(labels) if \_ == l])] = n

**print**('data',data.shape)

**print**('Labels and label indices', all\_labels)

import **scipy**

"""

函数 stft()：实现短时傅里叶变换STFT

输入参数 x：原始数据，ndarray；

        fftsize：FFT大小，默认为64

        overlap\_pct：重叠窗口大小(FFT大小的倍数)，默认为0.5

输出参数 raw[:,:(fftsize//2)]：二维STFT谱

"""

def **stft**(x, fftsize=64, overlap\_pct=.5):

    hop = **int**(fftsize \* (1 - overlap\_pct))

    w = **scipy**.hanning(fftsize + 1)[:-1] *# 转换为汉明窗口*

    raw = **np**.**array**([**np**.**fft**.**rfft**(w \* x[i:i + fftsize]) for i in **range**(0, **len**(x) - fftsize, hop)])

    return raw[:, :(fftsize // 2)]

from **numpy**.**lib**.**stride\_tricks** import **as\_strided**

"""

函数 peakfind()：STFT的峰值检测

输入参数 x：STFT谱；

        n\_peaks：最终选择峰值数

        l\_size：窗口左半部分大小，默认为3

        r\_size：窗口中间部分大小，默认为3

        c\_size：窗口右半部分大小，默认为3

        f：对每部分采用的统计函数，默认为均值np.mean

输出参数 heights：波峰对应的高度（幅值）

        top[:n\_peaks]：前n\_peaks个波峰的频率值

"""

def **peakfind**(x, n\_peaks, l\_size=3, r\_size=3, c\_size=3, f=**np**.**mean**):

    win\_size = l\_size + r\_size + c\_size

    shape = x.shape[:-1] + (x.shape[-1] - win\_size + 1, win\_size)

    strides = x.strides + (x.strides[-1],)

    xs = **as\_strided**(x, shape=shape, strides=strides)

    def **is\_peak**(x):

        centered = (**np**.**argmax**(x) == l\_size + **int**(c\_size/2))

        l = x[:l\_size]

        c = x[l\_size:l\_size + c\_size]

        r = x[-r\_size:]

        passes = **np**.**max**(c) > **np**.**max**([f(l), f(r)])

        if centered and passes:

            return **np**.**max**(c)

        else:

            return -1

    r = **np**.**apply\_along\_axis**(**is\_peak**, 1, xs)

    top = **np**.**argsort**(r, None)[::-1]

    heights = r[top[:n\_peaks]]

    top[top > -1] = top[top > -1] + l\_size + **int**(c\_size / 2.)

    return heights, top[:n\_peaks]

all\_obs = []

for i in **range**(data.shape[0]):

    d = **np**.abs(**stft**(data[i, :]))

    n\_dim = 6

    obs = **np**.**zeros**((n\_dim, d.shape[0]))

    for r in **range**(d.shape[0]):

        \_, t = **peakfind**(d[r, :], n\_peaks=n\_dim)

        obs[:, r] = t.copy()

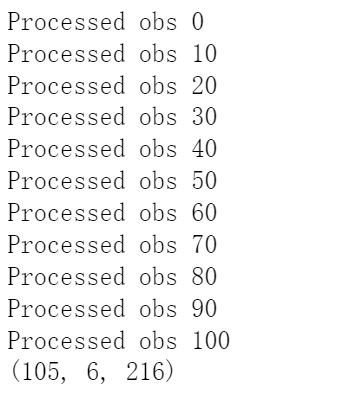
    if i % 10 == 0:

**print**("Processed obs %s" % i)

    all\_obs.**append**(obs)

all\_obs = **np**.**atleast\_3d**(all\_obs)

**print** (all\_obs.shape)



### 步骤二、GMM-HMM

1. GMM决定了隐马尔科夫模型中状态与输入语音帧之间的符合情况，HMM用来处理在时间轴上的声学可变性
2. HMM使用前后向算法

import **scipy**.**stats** as **st**

import **numpy** as **np**

class **GmmHmm**:

    def **\_\_init\_\_**(self, n\_states):

        self.n\_states = n\_states

        self.random\_state = **np**.**random**.**RandomState**(0)

*# 随机初始化*

        self.prior = self.**\_normalize**(self.random\_state.**rand**(self.n\_states, 1))

        self.A = self.**\_stochasticize**(self.random\_state.**rand**(self.n\_states, self.n\_states))

        self.mu = None

        self.covs = None

        self.n\_dims = None

    def **\_forward**(self, B):

        log\_likelihood = 0.

        T = B.shape[1]

        alpha = **np**.**zeros**(B.shape)

        for t in **range**(T):

            if t == 0:

                alpha[:, t] = B[:, t] \* self.prior.ravel()

            else:

                alpha[:, t] = B[:, t] \* **np**.**dot**(self.A.T, alpha[:, t - 1])

            alpha\_sum = **np**.**sum**(alpha[:, t])

            alpha[:, t] /= alpha\_sum

            log\_likelihood = log\_likelihood + **np**.log(alpha\_sum)

        return log\_likelihood, alpha

    def **\_backward**(self, B):

        T = B.shape[1]

        beta = **np**.**zeros**(B.shape);

        beta[:, -1] = **np**.**ones**(B.shape[0])

        for t in **range**(T - 1)[::-1]:

            beta[:, t] = **np**.**dot**(self.A, (B[:, t + 1] \* beta[:, t + 1]))

            beta[:, t] /= **np**.**sum**(beta[:, t])

        return beta

    def **\_state\_likelihood**(self, obs):

        obs = **np**.**atleast\_2d**(obs)

        B = **np**.**zeros**((self.n\_states, obs.shape[1]))

        for s in **range**(self.n\_states):

*#N 这一部分需要 0.14版本以上的Scipy*

**np**.**random**.seed(self.random\_state.**randint**(1))

            B[s, :] = **st**.multivariate\_normal.**pdf**(

                obs.T, mean=self.mu[:, s].T, cov=self.covs[:, :, s].T)

        return B

    def **\_normalize**(self, x):

        return (x + (x == 0)) / **np**.**sum**(x)

    def **\_stochasticize**(self, x):

        return (x + (x == 0)) / **np**.**sum**(x, axis=1)

    def **\_em\_init**(self, obs):

        if self.n\_dims is None:

            self.n\_dims = obs.shape[0]

        if self.mu is None:

            subset = self.random\_state.**choice**(**np**.**arange**(self.n\_dims), size=self.n\_states, replace=False)

            self.mu = obs[:, subset]

        if self.covs is None:

            self.covs = **np**.**zeros**((self.n\_dims, self.n\_dims, self.n\_states))

            self.covs += **np**.**diag**(**np**.**diag**(**np**.**cov**(obs)))[:, :, None]

        return self

    def **\_em\_step**(self, obs):

        obs = **np**.**atleast\_2d**(obs)

        B = self.**\_state\_likelihood**(obs)

        T = obs.shape[1]

        log\_likelihood, alpha = self.**\_forward**(B)

        beta = self.**\_backward**(B)

        xi\_sum = **np**.**zeros**((self.n\_states, self.n\_states))

        gamma = **np**.**zeros**((self.n\_states, T))

        for t in **range**(T - 1):

            partial\_sum = self.A \* **np**.**dot**(alpha[:, t], (beta[:, t] \* B[:, t + 1]).T)

            xi\_sum += self.**\_normalize**(partial\_sum)

            partial\_g = alpha[:, t] \* beta[:, t]

            gamma[:, t] = self.**\_normalize**(partial\_g)

        partial\_g = alpha[:, -1] \* beta[:, -1]

        gamma[:, -1] = self.**\_normalize**(partial\_g)

        expected\_prior = gamma[:, 0]

        expected\_A = self.**\_stochasticize**(xi\_sum)

        expected\_mu = **np**.**zeros**((self.n\_dims, self.n\_states))

        expected\_covs = **np**.**zeros**((self.n\_dims, self.n\_dims, self.n\_states))

        gamma\_state\_sum = **np**.**sum**(gamma, axis=1)

*#在除法运算之前，将0改为1*

        gamma\_state\_sum = gamma\_state\_sum + (gamma\_state\_sum == 0)

        for s in **range**(self.n\_states):

            gamma\_obs = obs \* gamma[s, :]

            expected\_mu[:, s] = **np**.**sum**(gamma\_obs, axis=1) / gamma\_state\_sum[s]

            partial\_covs = **np**.**dot**(gamma\_obs, obs.T) / gamma\_state\_sum[s] - **np**.**dot**(expected\_mu[:, s], expected\_mu[:, s].T)

*# 对称*

            partial\_covs = **np**.**triu**(partial\_covs) + **np**.**triu**(partial\_covs).T - **np**.**diag**(partial\_covs)

*#加上对角元素以保证半正定*

        expected\_covs += .01 \* **np**.**eye**(self.n\_dims)[:, :, None]

        self.prior = expected\_prior

        self.mu = expected\_mu

        self.covs = expected\_covs

        self.A = expected\_A

        return log\_likelihood

    def **fit**(self, obs, n\_iter=15):

*# obj支持2D或者3D数组*

*# 2D数组的形式为 n\_features, n\_dims*

*# 3D数组的形式为 n\_examples, n\_features, n\_dims*

*# 例如一个语音片段若包含6个特征，并且由105个不同的样本*

*# 那么这个obj数组为三维，大小为(105,6,X)，其中X为frames的个数*

*# 对于只包含一个样本的数据obj大小为(6,X)*

        if **len**(obs.shape) == 2:

            for i in **range**(n\_iter):

                self.**\_em\_init**(obs)

                log\_likelihood = self.**\_em\_step**(obs)

        elif **len**(obs.shape) == 3:

            count = obs.shape[0]

            for n in **range**(count):

                for i in **range**(n\_iter):

                    self.**\_em\_init**(obs[n, :, :])

                    log\_likelihood = self.**\_em\_step**(obs[n, :, :])

        return self

    def **transform**(self, obs):

*# obj支持2D或者3D数组*

*# 2D数组的形式为 n\_features, n\_dims*

*# 3D数组的形式为 n\_examples, n\_features, n\_dims*

*# 例如一个语音片段若包含6个特征，并且由105个不同的样本*

*# 那么这个obj数组为三维，大小为(105,6,X)，其中X为frames的个数*

*# 对于只包含一个样本的数据obj大小为(6,X)*

        if **len**(obs.shape) == 2:

            B = self.**\_state\_likelihood**(obs)

            log\_likelihood, \_ = self.**\_forward**(B)

            return log\_likelihood

        elif **len**(obs.shape) == 3:

            count = obs.shape[0]

            out = **np**.**zeros**((count,))

            for n in **range**(count):

                B = self.**\_state\_likelihood**(obs[n, :, :])

                log\_likelihood, \_ = self.**\_forward**(B)

                out[n] = log\_likelihood

            return out

from **sklearn**.**model\_selection** import **train\_test\_split**

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(all\_obs,

                                                    all\_labels,

                                                    train\_size=0.8,

                                                    test\_size=0.2,

                                                    random\_state =666,

                                                    stratify=all\_labels)

*# 由于HMM得到的结果是概率，所以我们需要对特征进行归一化*

for n,i in **enumerate**(all\_obs):

    all\_obs[n] /= all\_obs[n].sum(axis=0)

**print**('Size of training matrix:', X\_train.shape)

**print**('Size of testing matrix:', X\_test.shape)

## 结果分析

1. 结果并不是十分的理想

ys = **set**(all\_labels)

ms = [**GmmHmm**(6) for y in ys]

\_ = [m.**fit**(X\_train[y\_train == y, :, :]) for m, y in **zip**(ms, ys)]

ps = [m.**transform**(X\_test) for m in ms]

res = **np**.**vstack**(ps)

predicted\_labels = **np**.**argmax**(res, axis=0)

missed = (predicted\_labels != y\_test)

**print**('Test accuracy: %.2f percent' % (100 \* (1 - **np**.**mean**(missed))))

**Test accuracy: 14.29 percent**