基于 GMM 的语音独立词识别 实验报告

2018202177 官佳薇

一 实验内容

基于 0-9 数字语音数据集，使用 GMM 对 10 个数字逐一建模，对输入的音频进行分类，识别语音中表达的数字。

二 实验方案

1. MFCC 特征提取

使用包括 0-9 共十个数字的英文数据集，每个数字包含 20 条.wav 格式音频文件。使用两种方式获取 MFCC 特征，分别比较模型表现：

(1) Python-speech-features package

使用 python 中的 wave 包读取.wav 文件，并使用 python-speech-features 包对每条音频提取 13 维 mfcc 特征。同时对提取包含一、二阶导的 39 维 mfcc 特征进行了测试，但结果不如 13 维 mfcc 特征。

(2) Librosa package

使用 Librosa 自带库函数读取音频，并对每条音频提取 13 维 mfcc 特征。同时对包含一、二阶导的 39 维 mfcc 特征进行了测试，结果同样不如 13 维 mfcc 特征。使用 Librosa 提取特征代入 GMM 模型训练的效果整体优于 Python-speech-features 库。

2. 模型构建

1) GMM 模型

a) GMM 类定义

定义一个 GMM 类，其中包含参数初始化、设置最优参数、模型训练、获取分值四个主体部分。类定义如下：

```
class GMM:
    def __init__(self, max_iter=150, n_clusters=1, init_method='sample',
                 reg_covar = 1e-6, n_init = 1, tol = 1e-3, covariance_type='diag'):
        self.max_iter = max_iter
        self.converged = False
        self.init_method = init_method
        self.n_clusters = n_clusters
        self.reg_covar = reg_covar
        self.n_init = n_init
        self.tol = tol
        self.covariance_type = covariance_type
```

参数：

- max_iter：最大迭代次数，默认 150.
- n_clusters：混合高斯模型个数（即聚类个数），默认 1 个。
- init_method：初始化方法，可选 sample 方法（此处按样本数和 n_clusters 进行平均分割）和 kmeans 方法。
- reg_covar：正则化参数，以保证方差不为 0，默认 1e-6。
- n_init：执行初始化次数，默认为 1。
- tol：EM 迭代停止阈值，默认 1e-3。

• covariance_type：默认'diag'，本次实现算法中只支持'diag'方式，即使用协方差矩阵对角线元素进行算法求解。

b) Clusters 平均分初始化

对样本按 n_clusters 聚类个数进行平均分割，计算各类均值和方差(diagonal)，并按每个 cluster 中的样本个数计算权重。

```
def init_uniform(self, X):
    '''
    平均分初始化n_cluster个高斯分布
    返回各高斯分布的均值mean，diagonal方差variance，各部分比例prob，每个样本的类别号label
    '''
    featnum_per_group = int(np.ceil(X.shape[0]/self.n_clusters))
    mean = np.zeros((self.n_clusters, X[0].shape[0]))
    variance = np.zeros((self.n_clusters, X[0].shape[0]))   # diagonal variances
    prob = np.zeros(self.n_clusters)
    label = np.zeros(X.shape[0])
    for i in range(self.n_clusters):
        mean[i] = np.sum(X[i*featnum_per_group: (i+1)*featnum_per_group], axis=0)
                    / X[i*featnum_per_group: (i+1)*featnum_per_group].shape[0]
        variance[i] = np.sum((X[i*featnum_per_group: (i+1)*featnum_per_group] - mean[i]) *
                             (X[i*featnum_per_group: (i+1)*featnum_per_group] - mean[i]), axis=0) /
                             X[i*featnum_per_group: (i+1)*featnum_per_group].shape[0]
        prob[i] = X[i*featnum_per_group: (i+1)*featnum_per_group].shape[0] / X.shape[0]
        label[i*featnum_per_group: (i+1)*featnum_per_group] = i
    return mean, variance, prob, label
```

c) Clusters Kmeans 初始化

使用 sklearn 库中的 Kmeans 方法，将样本分成 n_clusters 个类，计算各类均值和方差(diagonal)，并按各类别中的样本个数计算权重。

```
def init_kmeans(self, X):
    '''
    输入样本X (n_samples, n_features)
    使用kmeans方法初始化均值、方差、权重
    '''
    label = KMeans(n_clusters=self.n_clusters, ).fit(X).labels_     # n_init=1
    mean = np.zeros((self.n_clusters, X[0].shape[0]))
    variance = np.zeros((self.n_clusters, X[0].shape[0]))   # diagonal
    prob = np.zeros(self.n_clusters)
    for i in range(self.n_clusters):
        seg = X[np.where(label == i)]
        mean[i] = np.sum(seg, axis=0) / seg.shape[0]
        variance[i] = np.sum((seg - mean[i]) * (seg - mean[i]), axis=0) / seg.shape[0] + self.reg_covar
        prob[i] = seg.shape[0] / X.shape[0]
    return mean, variance, prob, label
```

d) 模型训练（EM 过程）

对输入样本 X，使用 EM 方法迭代获得最优 GMM 模型。

首先调用上述方法对类别进行初始化，得到各类均值、方差、权重。

根据 EM 公式迭代计算，第一步计算各样本 xi 属于各类别 h 的高斯分布概率值 p(xi|h)。第二步根据高斯分布概率计算得到各样本 xi 属于各类别 h 的后验概率 p(h|xi)。最后根据后验概率计算得到各类新的均值、方差、类别概率，其中方差为 diagonal 方差，即计算协方差矩阵对角线元素。每一轮迭代中计算样本 xi 属于该 GMM 模型中各类 h 的后验概率之和的对数似然均值 lower_bound，函数表达如下：

$$\ln p(X|\pi,\mu,\Sigma) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathbb{N}(x_n|\mu_k,\Sigma_k) \right\}$$

若前后两次迭代的 lower_bound 值变化小于 tol，则认为模型收敛，跳出迭代，否则迭代至 max_iter 次。

对不同的 n_clusters，重复上述 EM 过程，根据 BIC 分值选取最佳 GMM 模型。

实现过程中使用 numpy 点积等矩阵运算代替了 for 循环逐一求解过程，将方差的计算转换为了矩阵运算，使运算速度大幅提升。

代码：

```
    def fit(self, X):
        X = np.array(X)
        for init in range(self.n_init):
            if self.init_method == 'sample':
                mean, variance, pre_prob, label = self.init_uniform(X)
            elif self.init_method == 'kmeans':
                mean, variance, pre_prob, label = self.init_kmeans(X)

            self.n_clusters = init + 1  # 类别数
            lower_bound = -np.infty
            max_lower_bound = -np.infty
            # 迭代
            for n_iter in range(self.max_iter):
                prev_lower_bound = lower_bound
                # E step
                # 计算每个样本属于各个高斯分布的p(xi|h)
                mean2 = np.sum(mean**2/variance, axis=1)[:, np.newaxis]
                X2 = np.dot(1/variance, (X**2).T)
                X_mean = np.dot(mean/variance, X.T)
                var = np.prod(np.sqrt(variance), axis=1)[:, np.newaxis]
                gauss_prod_matrix = np.exp(-.5 * (mean2 + X2 - 2*X_mean))
                                    / (np.power((2*np.pi), X[0].shape[0]/2) * var)
                # 求解p(h)*p(x|h)
                weighted_prob_matrix = pre_prob[:, np.newaxis] * gauss_prod_matrix  # p(h) * p(x|h)
                weighted_prob_sum = np.sum(weighted_prob_matrix, axis=0)

                # 本轮迭代分值
                lower_bound = np.mean(np.log(weighted_prob_sum))
```

```
                # M step
                pos_prob_matrix = weighted_prob_matrix / weighted_prob_sum  # 每个样本属于每个类的后验概率p(h|x)
                pos_prob_sum = (np.sum(pos_prob_matrix, axis=1)+
                                10 * np.finfo(pos_prob_matrix.dtype).eps)[:, np.newaxis]

                mean = np.dot(pos_prob_matrix, X) / pos_prob_sum    # 更新均值
                # 更新方差
                avg_X2 = np.dot(pos_prob_matrix, X * X) / pos_prob_sum
                avg_means2 = mean ** 2
                avg_X_means = mean * np.dot(pos_prob_matrix, X) / pos_prob_sum
                variance = avg_X2 - 2 * avg_X_means + avg_means2 + self.reg_covar
                # 更新权重（先验概率）
                pre_prob = np.sum(pos_prob_matrix, axis=1) / X.shape[0]

                # 判断是否收敛
                change = lower_bound - prev_lower_bound
                if change < self.tol:
                    self.converged = True
                    break
            # 保存最优模型参数
            if lower_bound > max_lower_bound:
                max_lower_bound = lower_bound
                best_params = (pre_prob, mean, variance)
                best_n_iter = n_iter
                best_n_clusters = self.n_clusters

        if not self.converged:
            print("Warning... Not converged...")
```

```
        if not self.converged:
            print("Warning... Not converged...")

        # 将模型参数设置为最优参数
        self._set_parameters(best_params)
        self.n_iter_ = best_n_iter
        self.lower_bound = max_lower_bound
        self.n_clusters = best_n_clusters

        print("Best parameters: n_clusters = {}, lower_bound = {}".format(self.n_clusters,
self.lower_bound))
```

e) 计算样本分值

对输入的样本 X，计算其在 GMM 模型中得到的分值。仍使用如下表达式进行分值计算，以评估样本属于该 GMM 模型的可能性。

$$\ln p(X|\pi,\mu,\Sigma) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathbb{N}(x_n|\mu_k,\Sigma_k) \right\}$$

代码：

```
    def score(self, X):
        # 计算各样本xi属于各类别h的高斯分布概率p(xi|h)
        mean2 = np.sum(self.mean ** 2 / self.variance, axis=1)[:, np.newaxis]
        X2 = np.dot(1 / self.variance, (X ** 2).T)
        X_mean = np.dot(self.mean / self.variance, X.T)
        var = np.prod(np.sqrt(self.variance), axis=1)[:, np.newaxis]
        gauss_prod_matrix = np.exp(-.5 * (mean2 + X2 - 2 * X_mean)) /
                                    (np.power((2 * np.pi), X[0].shape[0] / 2) * var)
        # 计算p(h)*p(x|h)
        weighted_prob_matrix = self.pre_prob[:, np.newaxis] * gauss_prod_matrix  # p(h) * p(x|h)
        return np.mean(np.log(np.sum(weighted_prob_matrix, axis=0)))
```

2) GMM_Set 模型构建

对 0-9 共 10 个数字的 10 个 GMM 模型统一封装成 GMM_Set 模型。

a) GMM_Set 类定义

维护 gmm_models 数组，用于存放 0-9 共 10 个 GMM 模型。维护 label 数组存放 GMM 模型对应的数字。

```
class GMM_Set:
    def __init__(self, max_iter = 150):
        self.gmm_models = []    # 0-9 one model for one number
        self.label = [] # label
        self.max_iter = max_iter
```

b) 添加 GMM 模型

传入参数 label 为数字编号，features 为该数字下的所有训练样本。定义并训练 GMM 模型，将模型添加至模型列表中。

```
    def add_new_model(self, features, label):
        self.label.append(label)    # 记录类别
        gmm = GMM()                 # 定义GMM模型
        gmm.fit(features)           # 模型训练
        self.gmm_models.append(gmm) # 加入模型集合
```

c) 数值预测

对每个模型调用 GMM 类中 score 函数获取分值，取值最大的下标作为预测的数字。

```
    def predict_num(self, input):
        scores = [self.cal_gmm_score(gmm, input) for gmm in self.gmm_models]  # 各模型分值
        result = [(self.label[index], value) for (index, value) in enumerate(scores)]
        p = max(result, key=operator.itemgetter(1))
        return p[0]
```

3) 封装模型，添加交互函数

GMM_Model 类包含 GMM_Set 模型组，训练数据 features 为 defaultdict 类型。类定义如下：

```
class GMM_Model:
    def __init__(self, features):
        self.features = features    # 特征
        self.gmms = GMM_Set()       # GMM模型组
```

包含模型训练、预测、模型保存(dump)和模型加载(load)函数。

3. 端点检测（对比实验中使用）

为提升模型准确性，考虑加入端点检测对音频进行预处理。利用短时能量、短时过零率和双门限法进行端点检测。

使用 wave 库读取音频文件，将音频数据根据采样点数转化为可计算的数组形式，在此基础上进行端点检测。定义 EndPointDetect 类，其中包含能量计算、过零率计算和端点检测方法。以 256 个采样点为一帧计算每一帧的能量、过零率，并按帧使用双门限法进行端点检测。

(1) 计算短时能量

短时能量是语音的时域特征，因此，在不使用傅里叶变换的情况下，这里的窗口是一种方窗。这里的语音短时能量就相当于，每一帧中所有语音信号的平方和。

```
@staticmethod
def calEnergy(wave_data) :
    energy = []
    sum = 0
    for i in range(len(wave_data)) :
        sum = sum + (int(wave_data[i]) * int(wave_data[i]))
        if (i + 1) % 256 == 0 :
            energy.append(sum)
            sum = 0
        elif i == len(wave_data) - 1 :
            energy.append(sum)
    return energy
```

(2) 计算短时过零率

短时过零率就是单位时间穿过坐标系横轴的次数。

```
@staticmethod
def calZeroCrossingRate(wave_data) :
    zeroCrossingRate = []
    sum = 0
    for i in range(len(wave_data)) :
        if i == 0:
            None
        sum = sum + np.abs(sgn(wave_data[i]) - sgn(wave_data[i - 1]))
        if (i + 1) % 256 == 0 :
            zeroCrossingRate.append(float(sum) / 255)
            sum = 0
        elif i == len(wave_data) - 1 :
            zeroCrossingRate.append(float(sum) / 255)
    return zeroCrossingRate
```

(3) 双门限法端点检测

根据前述得到的各帧短时能量和短时过零率，使用经验函数求得较高和较低 2 个能量阈值，和过零率阈值。采用双门限法，首先利用较大能量阈值，选出能量大于该阈值的音频前后端点。再根据较低能量阈值，扩张端点。最后利用过零率做最后一步检测。

```
@staticmethod
def endPointDetect(wave_data, energy, zeroCrossingRate) :
    sum = 0
    energyAverage = 0
    for en in energy:
        sum = sum + en
    energyAverage = sum / len(energy)

    sum = 0
    for en in energy[:5] :
        sum = sum + en
    ML = sum / 5
    MH = energyAverage / 4       #较高的能量阈值
    ML = (ML + MH) / 4     #较低的能量阈值
    sum = 0
    for zcr in zeroCrossingRate[:5] :
        sum = float(sum) + zcr
    Zs = sum / 5                        #过零率阈值
```

```
    A = []
    B = []
    C = []

    # 首先利用较大能量阈值 MH 进行初步检测，记录端点
    flag = 0    # 记录A数组下标%2的余数
    for i in range(len(energy)):
        # 首次遇到大于MH的元素，记录下标
        if len(A) == 0 and flag == 0 and energy[i] > MH :
            A.append(i)
            flag = 1
        elif flag == 0 and energy[i] > MH and i - 21 > A[len(A) - 1]:
            A.append(i)
            flag = 1
        elif flag == 0 and energy[i] > MH and i - 21 <= A[len(A) - 1]:  # 过短，删除
            A = A[:len(A) - 1]
            flag = 1

        # 大于阈值MH的范围结束
        if flag == 1 and energy[i] < MH :
            # 检测帧长  如果超过MH的帧长太短，那就去掉
            if i - A[len(A) - 1] <= 2 :
                A = A[:len(A) - 1]
            else :
                A.append(i)
            flag = 0
    # print("较高能量阈值，计算后的浊音A:" + str(A))
```

```
    # 利用较小能量阈值 ML 进行第二步能量检测 扩张端点
    for j in range(len(A)) :
        i = A[j]
        if j % 2 == 1 :
            while i < len(energy) and energy[i] > ML :
                i = i + 1
            B.append(i)
        else :
            while i > 0 and energy[i] > ML :
                i = i - 1
            B.append(i)
    # print("较低能量阈值，增加一段语言B:" + str(B))

    # 利用过零率进行最后一步检测 扩张端点
    for j in range(len(B)) :
        i = B[j]
        if j % 2 == 1 :
            while i < len(zeroCrossingRate) and zeroCrossingRate[i] >= 3 * Zs :
                i = i + 1
            C.append(i)
        else :
            while i > 0 and zeroCrossingRate[i] >= 3 * Zs :
                i = i - 1
            C.append(i)
    # print("过零率阈值，最终语音分段C:" + str(C))
    return C
```

三 实验结果

1. 模型构建时间
   由于在 EM 迭代过程中有效利用 numpy 矩阵运算代替了 for 循环，模型构建用时较短，用时 0.02 秒。

```
Best parameters: n_clusters = 1, lower_bound = -138793.7349412041
covariance_type = diag
Best parameters: n_clusters = 1, lower_bound = -138948.97239530744
Model Training Time : 0.022939205169677734 seconds
```

2. Librosa 提取 39 维 mfcc 特征+各数字 15 条训练 5 条测试
   准确率 42%

```
label, pred = (8, 8)
label, pred = (9, 7)
label, pred = (9, 7)
label, pred = (9, 7)
label, pred = (9, 9)
label, pred = (9, 7)
precision: 0.420000
```

3. Librosa 提取 13 维 mfcc 特征+各数字 15 条训练 5 条测试
   准确率 58%

```
label, pred = (8, 8)
label, pred = (9, 1)
label, pred = (9, 7)
label, pred = (9, 9)
label, pred = (9, 7)
label, pred = (9, 7)
precision: 0.580000
```

4. Librosa 提取 13 维 mfcc 特征+各数字 10 条训练 5 条测试
   准确率 58%

```
label, pred = (9, 9)
label, pred = (9, 1)
label, pred = (9, 7)
label, pred = (9, 9)
label, pred = (9, 9)
label, pred = (9, 9)
precision: 0.580000
```

5. 端点检测+librosa 提取 13 维 mfcc 特征+各数字 15 条训练 5 条测试
   准确率 56%

```
label, pred = (8, 9)
label, pred = (9, 5)
label, pred = (9, 1)
label, pred = (9, 5)
label, pred = (9, 1)
label, pred = (9, 1)
precision: 0.560000
```

6. 使用 Python speech features 提取 mfcc 特征得到的模型准确率较低，故不冗余展示。

四 使用 Scikit-learn 混合高斯模型进行比较

使用 scikit-learn 的 GaussianMixture 混合高斯模型对 0-9 每个数字各构建一个单核 GMM。对每个数字根据一定比例划分出来的训练样本，使用这部分训练数据训练该数字的单核 GMM 模型。

Scikit-learn 中对 GaussianMixture 模型的定义如下：

```
class sklearn.mixture.GaussianMixture(n_components=1, covariance_type='full', tol=0.001,
                                       reg_covar=1e-06, max_iter=100, n_init=1,
                                       init_params='kmeans', weights_init=None,
                                       means_init=None, precisions_init=None,
                                       random_state=None, warm_start=False, verbose=0,
                                       verbose_interval=10)
```

参数：

- n_components：混合高斯模型个数，默认为 1。
- covariance_type：描述要使用的协方差参数类型的字符串，必选一个{'full', 'tied', 'diag', 'spherical'}，默认为 full。full:每个混合元素有它公用的协方差矩阵；tied:每个混合元素共享同一个公共的协方差矩阵；diag:每个混合元素有它自己的对角矩阵；spherical:每个混合元素都有自己单独的方差值。
- tol：float 类型，EM 迭代停止阈值，默认值 1e-3。
- reg_covar：float 类型，协方差对角线上的非负正则化参数，默认值 1e-6,接近于 0。
- max_iter：最大迭代次数，默认为 100。
- n_init：执行初始化操作数量，保持最好的结果，默认为 1。
- init_params：可选 {'kmeans', 'random'}，默认值为 'kmeans'。初始化权重、均值及精度的方法，作用：用随机方法还是用 kmeans 方法初始化。
- weights_init：各组成模型的先验权重，可以自己设置，more 按照 7)产生。
- means_init：初始化均值，同 8)。
- precisions_init：初始化精确度（模型个数、特征个数），默认按照 7)实现。
- random_state：随机计数发生器。
- warm_start：若为 True，则 fit()调用会以上一次 fit()的结果作为初始化参数，适合相同问题多次 fit 的情况，能加速收敛，默认为 False。
- verbose：使能迭代信息显示，默认为 0，可以为 1 或大于 1（显示的信息不同）。
- verbose_interval：与 13)相关，若使能迭代信息显示，设置多少次迭代后显示信息，默认 10 次。

实验结果

1. 提取 13 维 mfcc 特征，每个数字前 15 条训练后 5 条测试，covariance_type='full'时准确率 76%，covariance_type='diag'时，准确率同样为 58%。

```
label, pred = (8, 8)
label, pred = (8, 8)
label, pred = (9, 9)
label, pred = (9, 9)
label, pred = (9, 9)
label, pred = (9, 9)
label, pred = (9, 1)
precision: 0.760000
```

```
label, pred = (8, 8)
label, pred = (8, 8)
label, pred = (9, 1)
label, pred = (9, 7)
label, pred = (9, 9)
label, pred = (9, 7)
label, pred = (9, 7)
precision: 0.580000
```

2. 提取含 1、2 阶导的 39 维 mfcc 特征，每个数字前 15 条训练后 5 条测试，covariance_type='full'准确率为 42%，covariance_type='diag'准确率为 42%。

```
label, pred = (8, 8)
label, pred = (9, 4)
label, pred = (9, 5)
label, pred = (9, 5)
label, pred = (9, 5)
label, pred = (9, 9)
precision: 0.420000
```

```
label, pred = (8, 8)
label, pred = (9, 7)
label, pred = (9, 7)
label, pred = (9, 7)
label, pred = (9, 9)
label, pred = (9, 7)
precision: 0.420000
```

3. 加入端点检测，提取 13 维 mfcc 特征，各数字前 15 条训练后 5 条测试，covariance_type='full'准确率 68%，covariance_type='diag'准确率 56%。

```
label, pred = (8, 8)
label, pred = (9, 9)
label, pred = (9, 9)
label, pred = (9, 5)
label, pred = (9, 1)
label, pred = (9, 1)
precision: 0.680000
```

```
label, pred = (8, 9)
label, pred = (9, 5)
label, pred = (9, 1)
label, pred = (9, 5)
label, pred = (9, 1)
label, pred = (9, 1)
precision: 0.560000
```

五 实验结果总结分析

本次实验使用了两种 mfcc 特征提取方法，在 GMM 模型实现过程中通过类封装、numpy 矩阵运算提高了模型构建效率，同时加入了端点检测进行测试。与 sklearn 库函数 covariance_type='diag'的情况进行比较，得到准确率基本一致。总体而言，使用 librosa 库提取 mfcc 特征构建 GMM 模型的效果优于 python speech features 库，且 13 维的 mfcc 特征效果比 39 维 mfcc 特征更好，考虑原因为训练数据过少导致的过拟合。同时方差计算使用 diagonal 的方式准确率不如使用协方差矩阵。加入端点检测对结果总体有提升，准确率不同程度增高。

模型实现过程中对算法进行了多处优化，以提高模型构建效率，最终模型构建用时 0.02 秒。实验准确率从 42%~58%不等。其中以 librosa 库提取的 13 维 mfcc 特征为基础准确率较高。

六 实验问题及解决

1. 方差为 0。在 EM 算法迭代过程中，某些类中样本数逐渐减少，最终导致方差为 0，使得结果全部取值 nan。
   解决：加入正则化参数 reg_covar，默认等于 1e-6。在每一轮更新方差时将协方差对角线各元素加上正则化参数，使算法正常运行。

2. 训练效率。EM 迭代计算时，由于需要计算每个样本 xi 属于每个高斯分布 h 的高斯分布概率 p(xi | h)，后验概率 p(h | xi)，且需要用该值计算均值、方差、类权重，使用 for 循环对每个高斯分布分别计算时间复杂度较高，耗时较长。
   解决：使用矩阵点积运算化简。例如，原始高斯分布密度概率计算公式如下（diagonal）：

$$p(xi|h) = \frac{1}{(2\pi)^{d/2}\prod_i \sigma_i} e^{-\frac{1}{2}\sum_i^d \frac{(x_i-\mu_i)^2}{\sigma_i^2}}$$

   由于输入的矩阵 X 大小为(n_samples, d)，各分布均值矩阵 mean 和方差矩阵 variance 大小为(n_clusters, d)，故考虑将公式稍作调整：

$$p(x_i|h) = \frac{1}{(2\pi)^{d/2}\prod_i \sigma_i} e^{-\frac{1}{2}\sum_{i=1}^d \left(\frac{x_i^2}{\sigma_i^2} - 2\frac{x_i\mu_i}{\sigma_i^2} + \frac{\mu_i^2}{\sigma_i^2}\right)}$$

   从而满足矩阵运算，代码：

```
mean2 = np.sum(mean**2/variance, axis=1)[:, np.newaxis] # mean**2 / variance
X2 = np.dot(1/variance, (X**2).T)                       # X**2 / variance
X_mean = np.dot(mean/variance, X.T)                     # mean * X / variance
var = np.prod(np.sqrt(variance), axis=1)[:, np.newaxis]
gauss_prod_matrix = np.exp(-.5 * (mean2 + X2 - 2*X_mean)) / (np.power((2*np.pi), X[0].shape[0]/2) * var)
```

四 使用手册

运行 endpoint_audio.py 可对 records 文件夹下所有文件进行端点检测，并生成相应处理后的文件存储在./Processed_records 文件夹下。

模型训练和预测默认以当前最优模型组合进行，即以 Librosa 提取 13 维 mfcc 特征构建 GMM。

```
python endpoint_audio.py    # 生成端点检测后的音频，默认使用records文件夹下数据
python Recognizer.py -t train -i "records/*/" -m model.out  # 训练模型
python Recognizer.py -t predict -i "records/*/" -m model.out  # 预测
```