哈尔滨工业大学计算学部

实验报告

课程名称： 机器学习

课程类型： 选修

实验题目：实现 k-means 聚类方法和混合高斯模型

学号：1190201215
姓名： 冯开来

一、实验目的

实现一个 k-means 算法和混合高斯模型，并且用 EM 算法估计模型中的参数。

二、实验要求及实验环境

2.1 实验要求

1. 用高斯分布产生 k 个高斯分布的数据（不同均值和方差）（其中参数自己设定）。
   a) 用 k-means 聚类，测试效果；
   b) 用混合高斯模型和你实现的 EM 算法估计参数，看看每次迭代后似然值变化情况，考察 EM 算法是否可以获得正确的结果（与你设定的结果比较）。
2. 应用：可以 UCI 上找一个简单问题数据，用你实现的 GMM 进行聚类。

2.2 实验环境

Windows10; python3.9;PyCharm 2021.2.2

三、设计思想（本程序中的用到的主要算法及数据结构）

3.1 EM 算法

本次实验主要分两部分进行。两个算法 K-means 和 GMM 本质是为 EM 算法的应用。所以我们先介绍 EM 算法。

EM 算法我看了很多文章和博客，到现在还是一知半解。简单来说就是分为 E 步和 M 步。E 步求的是期望（隐变量的概率分布），M 步就是求让这个期望最大的参数。

E 是调整分布，M 是根据调整的分布，求使得目标函数最大化的参数，从而更新了参数之后，又可以调整分布，直至收敛也就是参数不再有大的变化为止。

EM 算法对于初值很敏感，迭代过程中目标函数是向更优的方向趋近的，但是在某些情况下会陷入局部最优而非全局最优的问题。

3.2 K-means

方法 k-means 聚类就是根据某种度量方式(常用欧氏距离,如欧氏距离越小，相关性越大)，将相关性较大的一些样本点聚集在一起，一共聚成 k 个堆，每一个堆我们称为一“类”。k-means 的过程为：先在样本点中选取 k 个点作为暂时的聚类中心，然后依次计算每一个样本点与这 k 个点的距离，将每一个与距离这个点最近的中心点聚在一起，这样形成 k 个类“堆”，求每一个类的期望，将求得的期望作为这个类的新的中心点。一直不停地将所有样本点分为 k 类，直至中心点不再改变停止。

3.3 GMM

混合高斯模型指具有如下形式的概率分布模型：

$$P(y|\theta)=\sum_{k=1}^{K}\alpha_k\varphi(y|\theta_k)$$

其中 α k 是样本中类 k 的数据所占比例，$\sum_{k=1}^{K}\alpha_k=1$,$\varphi(y|\theta_k)$ 是第 k 类中高斯分布的概率分布函数。

其中$\varphi(y|\theta_k)$具体为

$$\varphi(y|\theta_k)=\frac{1}{2\pi^{\frac{D}{2}}|\Sigma_k|^{\frac{1}{2}}}exp(-\frac{(y-\mu_k)^T\Sigma_k^{-1}(y-\mu_k)}{2})$$

其中将(α,Σ,μ)记为θ，就是我们常说的隐变量，因为有隐变量，混合高斯模型无法求出解析解，但是可以用 EM 算法迭代求解完成分类。

具体的 EM 算法为：

1. 初始化响应度矩阵 γ，协方差矩阵，均值和 α
2. E 步：初始化响应度矩阵 γ，其中 γ jk 表示第 j 个样本属于第 k 类的概率，如下：

$$\gamma_{jk}=\frac{\alpha_k\varphi(y_j|\theta_k)}{\sum_{k=1}^{K}\alpha_k\varphi(y_j|\theta_k)},j=1,2,...,N;k=1,2,...,K$$

3. M 步：将响应度矩阵求解，更新均值，协方差矩阵和 α

$$\mu_k=\frac{\sum_{k=1}^{N}\gamma_{jk}y_j}{\sum_{j=1}^{N}N\gamma_{jk}},k=1,2,...,K$$

$$\Sigma_k=\frac{\sum_{j=1}^{N}\gamma_{jk}(y-\mu_k)(y-\mu_k)^T}{\sum_{j=1}^{N}\gamma_{jk}},k=1,2,..,K$$

4. 重复 2，3 步，迭代求解，直至 μ 的改变收敛。

3.4 算法实现

3.4.1 生成数据

首先初始化 k 个类和 count 个维度，然后通过输入的均值得到混合高斯模型。

```
def generate_data(k, count):
    means = np.zeros((k, 2))                    # 各个高斯分布的均值
    _data = np.zeros((count, 3))                  # 待分类的数据
    for i in range(k):
        means[i, 0] = float(input("输入第 " + str(i + 1) + " 组数据的均值\n"))
        means[i, 1] = float(input())
    cov = np.array([[1, 0], [0, 1]])              # 协方差矩阵
    index = 0
    for i in range(k):
        for j in range(int(count / k)):
            _data[index, 0:2] = np.random.multivariate_normal(means[i], cov)
            _data[index, 2] = i
            index += 1
    return means, _data
```

3.4.2 K-means

首先初始化 k 个类的中心，随机选取这 k 个样本中心，然后开始 K-means 迭代。通过计算每个样本到 k 个中心的距离（这里是欧式距离），然后选取最小的距离的那个 k，将该样本划分到这个类中，计算完所有的样本后得到 k 个类，然后重新计算类中心（均值），然后迭代，直至收敛。

```
def k_means(sample, num):                                    Reader Mode
    cnt= -1                              # 记录迭代次数
    dimension = sample.shape[1]                  # 数据的维度
    all_list = []                                        # 存放k个簇
    center = np.zeros((num, dimension))              # 记录k个簇的中心
    maxi = len(sample)
    rand = np.random.randint(0, maxi, num)       # 随机选取k个样本初始化
    for i in range(num):
        temp_list = [rand[i]]
        all_list.append(temp_list)
    while True:                                  # 迭代
        cnt += 1
        old_center = center.copy()               # 记录更新之前的中心
        for i in range(num):
            cluster = all_list[i]
            length = len(cluster)
            sums = np.zeros((1, dimension))      # 记录一个簇所有数据各维度的加和
            for j in range(dimension):           # 计算每个簇的中心
                for f in range(length):
                    sums[0, j] += sample[cluster[f], j]
                center[i, j] = float(sums[0, j] / length)  # 得到中心的各个维度
        if np.sum(abs(center - old_center)) < 0.1:       # center基本不变，结束迭代
            print(cnt)
            return center, all_list
        for i in range(num):
            all_list[i].clear()
        index = -1
        for info in sample:
            index += 1
            distance = np.zeros((1, num))  # 记录一个数据到三个中心的距离
            for i in range(num):
                for j in range(dimension):
                    distance[0, i] += float((center[i, j] - info[j]) ** 2)
            category = get_min(distance)             # 找到距离该数据最近的中心
```

其中得到距离样本最近的中心的函数 get_min()

```
# 找到最小距离
def get_min(distance):
    value = distance[0, 0]
    index = 0
    for i in range(1, distance.shape[1]):
        if value > distance[0, i]:
            value = distance[0, i]
            index = i
    return index
```

3.4.3 GMM

```python
# EM算法
def em_algorithm(center, sample, k):
    u, alpha, cov = init(center, sample, k)   # 初始化参数
    iterator = 0                              # 记录迭代次数
    while True:
        iterator += 1                         # 迭代次数
        prev_u = u.copy()                     # 记录上一次迭代的参数
        # prev_alpha = alpha.copy()
        # prev_cov = cov.copy()
        gama = gaussian_mixture(sample, k, u, alpha, cov)     # E步

        u, cov, alpha = like_hood(sample, gama, k)                  # M步
        if np.sum(abs(prev_u - u)) < 0.05:  # 均值基本不变，结束迭代
            print(iterator - 1)
            break
    cluster = classify(sample, k, u, cov, alpha)
    return u, cov, alpha, cluster
```

首先初始化参数响应度矩阵γ，协方差矩阵，均值和α。

```python
# 初始化先验概率、均值和协方差矩阵
def init(center, sample, num):
    dimension = sample.shape[1]
    # u = np.zeros((num, dimension))        # 初始化均值
    u = center.copy()
    covariance = np.eye(dimension)     # 初始化协方差
    cov = []
    alpha = np.zeros((num, 1))           # 初始化先验
    # maxi = len(sample)
    # rand = np.random.randint(0, maxi, num)  # 随机选取k个样本初始化
    for i in range(num):
        # u[i, :] = sample[rand[i]]
        alpha[i, 0] = float(1 / num)
        cov.append(covariance)
    return u, alpha, cov
```

然后开始迭代计算，首先是 E 步得到期望。

```python
# 通过均值、协方差矩阵和先验概率求得后验概率
def gaussian_mixture(sample, num, u, alpha, cov):
    number = sample.shape[0]
    gama = np.zeros((number, num))
    total_probability = []                          # 全概率
    for j in range(number):
        sums = 0
        for i in range(num):
            sums += float(alpha[i, 0] * gaussian_probability(sample[j, :], u[i, :], cov[i], num))
        total_probability.append(sums)              # 计算全概率

    for j in range(number):
        for i in range(num):                        # 计算xj属于i类的后验概率
            gama[j, i] = float(alpha[i, 0] * gaussian_probability(sample[j, :], u[i, :], cov[i], num)
                               / total_probability[j])
    return gama
```

其中有一个计算全概率公式的函数，是通过贝叶斯公式得到的：

```
def gaussian_probability(x, u, covariane, num):
    delta = x.reshape(len(x), 1) - u.reshape(len(u), 1)
    covariane1 = pow(np.linalg.det(covariane), 0.5)
    index1 = -num / 2
    pai = pow((2 * np.pi), index1)
    index2 = (-1/2) * np.dot(delta.T, np.linalg.inv(covariane)).dot(delta)
    prior = (covariane1 * pai * np.exp(index2))
    return prior
```

然后是 M 步，最大似然化期望。得到了新的 γ 然后用 γ 更新隐变量 θ。

```
def like_hood(sample, gama, num):
    dimension = sample.shape[1]
    number = sample.shape[0]
    u = np.zeros((num, dimension))      # 更新均值
    cov = []                            # 更新协方差矩阵
    alpha = np.zeros((num, 1))          # 更新先验
    # 更新均值
    for i in range(num):
        for j in range(dimension):
            divisor = 0
            dividend = 0
            for f in range(number):
                divisor += gama[f, i]
                dividend += gama[f, i] * sample[f, j]
            u[i, j] = float(dividend / divisor)
    # 更新协方差矩阵
    for i in range(num):
        divisor = 0
        dividend = np.zeros((dimension, dimension))
        for j in range(number):
            delta = sample[j].reshape(dimension, 1) - u[i, :].reshape(dimension, 1)
            matrix = np.dot(delta, delta.T)
            dividend += gama[j, i] * matrix
            divisor += gama[j, i]
        covariance = dividend / divisor
        cov.append(covariance)
    # 更新先验概率
    for i in range(num):
        gama_sum = 0
        for j in range(number):
            gama_sum += gama[j, i]
        alpha[i, 0] = float(gama_sum / number)
    return u, cov, alpha
```

开始迭代，直至均值的改变收敛于很小的数，那么可以停止迭代。得到结果：

```
    if np.sum(abs(prev_u - u)) < 0.05:  # 均值基本不变，结束迭代
        print(iterator - 1)
        break
cluster = classify(sample, k, u, cov, alpha)
return u, cov, alpha, cluster
```

3.4.4 绘图

```
def draw_point(point_set, num, center, accurate):
    count = int(len(point_set) / num)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("accuracy = " + str(accurate))
    for i in range(num):
        up = (i + 1) * count
        down = i * count
        set_x = point_set[down: up, 0]
        set_y = point_set[down: up, 1]
        plt.plot(set_x, set_y, linestyle='', marker='.')
    for i in range(num):
        plt.plot(center[i, 0], center[i, 1], linestyle='', marker='+', color='Black')
    plt.show()
```

3.4.5 准确率分析

将我们原先设置的分类和我们用 EM 算法的得到的分类进行比较和计算。

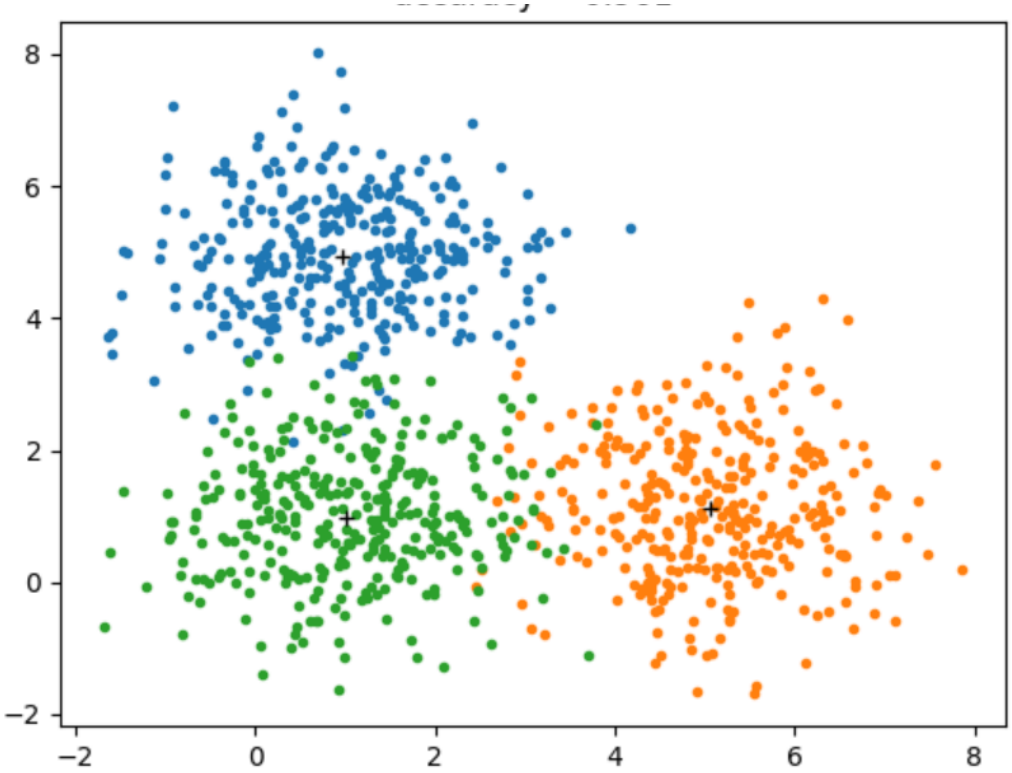
```
# 准确率分析
def accuracy(cluster, labels, num):
    sum_classified = 0
    for i in range(k):
        label_set = np.zeros((num, 1))
        for index in cluster[i]:
            index1 = int(labels[index])
            label_set[index1, 0] += 1
        sum_classified += get_attribute(label_set)
    my_accuracy = float(sum_classified / labels.shape[0])
    return my_accuracy
```

3.4.6 UCI 数据

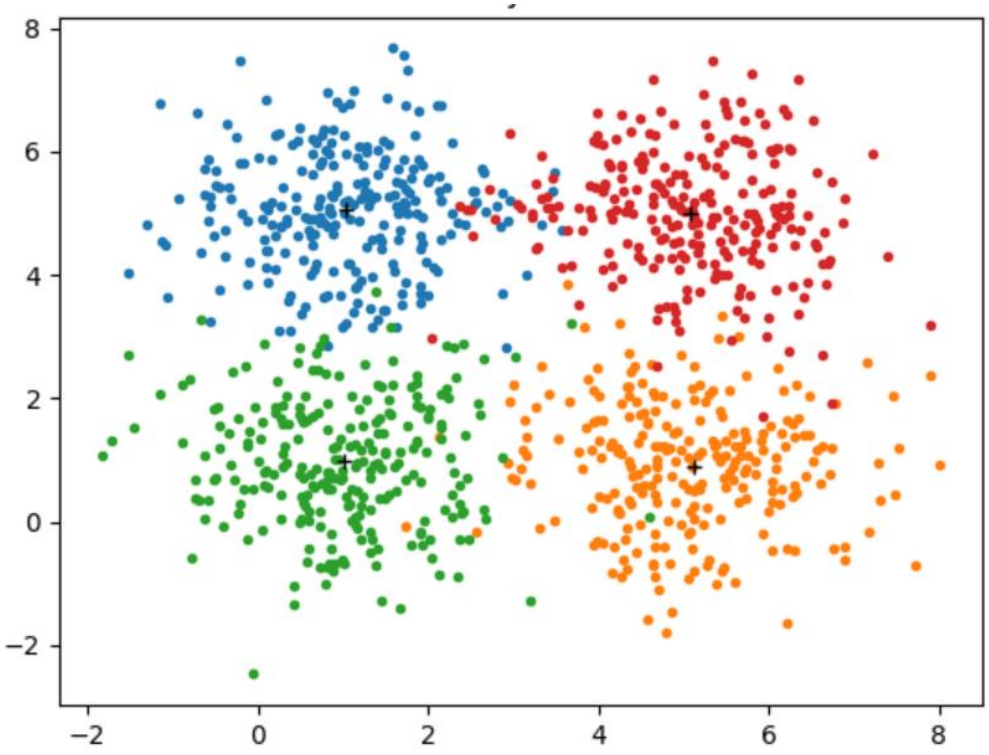
```
def get_uci():
    path = "Exasens.csv"
    data_set = pd.read_csv(path)
    x = data_set['Diagnosis']
    y = data_set.drop('Diagnosis', axis=1)
    _label, _data = np.array(x, dtype=str), np.array(y, dtype=int)
    for i in range(_label.shape[0]):
        if _label[i] == "COPD":
            _label[i] = 0
        elif _label[i] == "HC":
            _label[i] = 1
        elif _label[i] == "Asthma":
            _label[i] = 2
        elif _label[i] == "Infected":
            _label[i] = 3
    labels = np.array(_label, dtype=int)
    print()
    return labels, _data
```

四、实验结果与分析

4.1 生成的数据



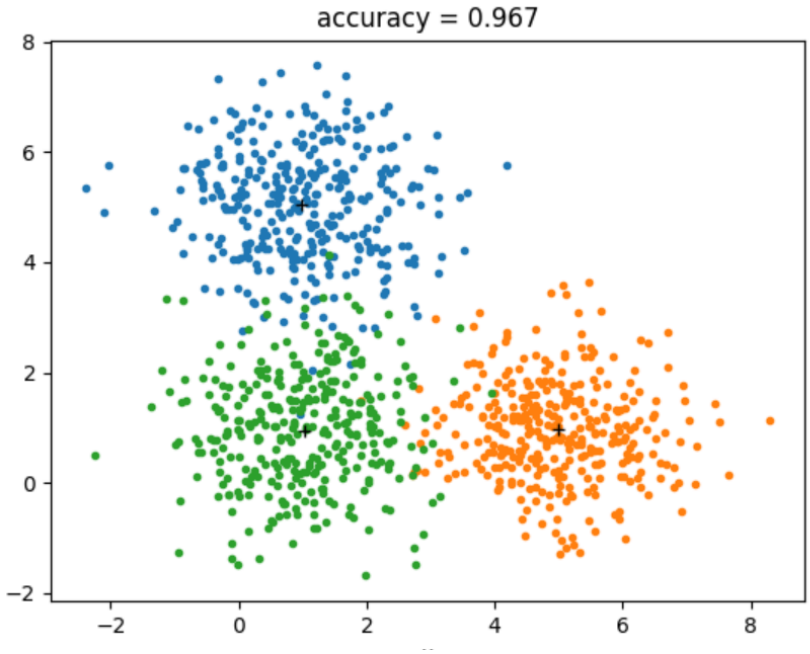
（三个类，中心分别为（1，5） （5，1） （1，1））



（四个类，中心分别为（1，5） （5，1） （1，1） （5，5））

4.2 K-means

（三个类，中心分别为（1，5） （5，1） （1，1））

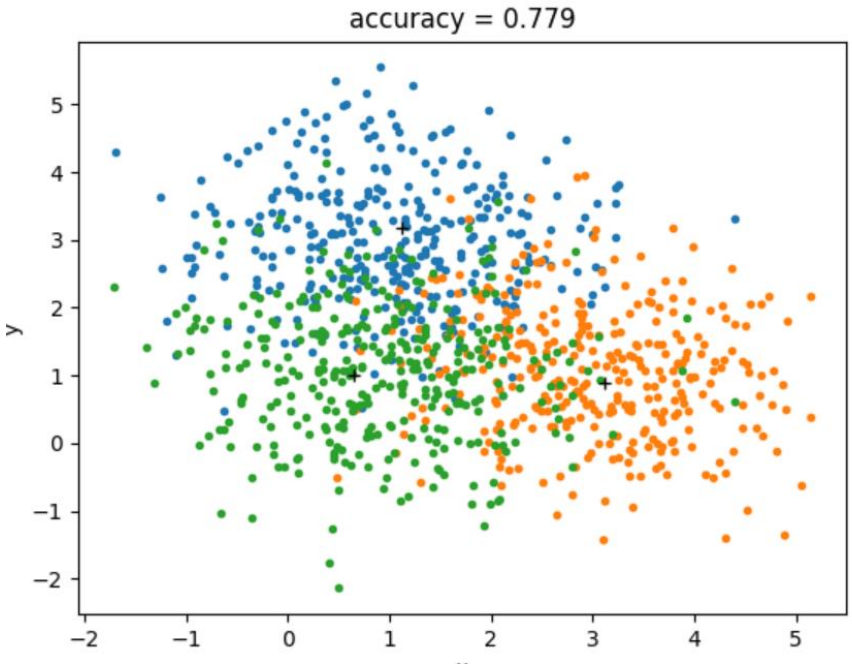


准确率 96.7%

得到的中心（可以看到还是很接近的，效果还不错）

```
[[0.96009298 0.91957469]
 [1.05062138 5.05678588]
 [4.97131559 0.99898784]]
```

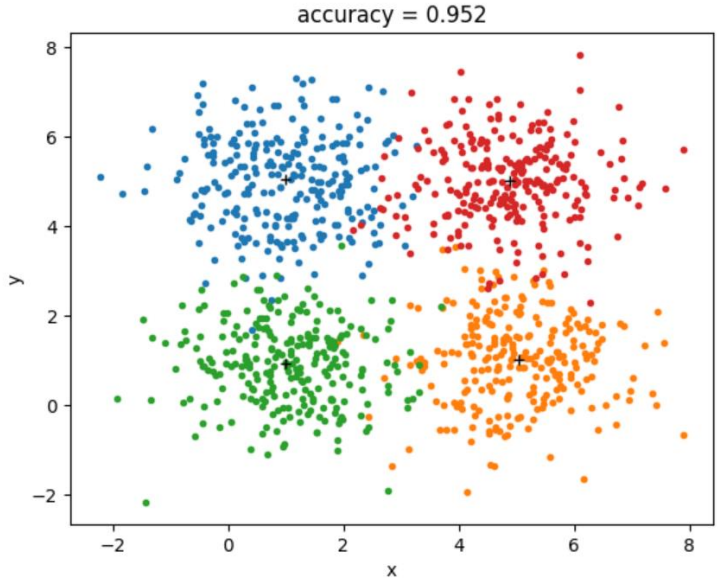
（三个类，中心分别为（1，3） （3，1） （1，1））



准确率 77.9%（因为样本就很难分清，属于贝叶斯错误率）
得到的中心（可以看到还是很接近的，效果还不错）

```
[[3.11394637 0.89142518]
 [0.64543876 1.01333299]
 [1.12823735 3.17177117]]
```

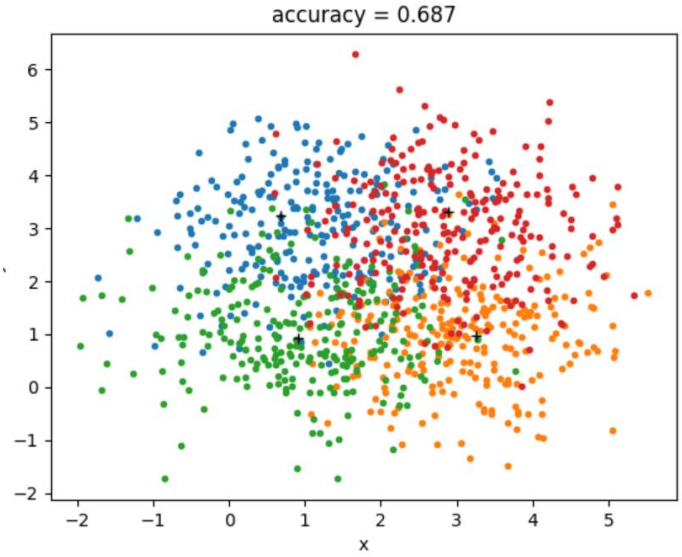
（四个类，中心分别为（1，5） （5，1） （1，1） （5，5））



准确率 95.2%
得到的中心（比三个类稍微差一点，但是准确率依旧很高）

```
[[5.03346579 1.02366579]
 [4.89012779 5.03205601]
 [0.9912728  0.95425014]
 [0.9967389  5.04906933]]
```

（四个类，中心分别为（1，5） （5，1） （1，1） （5，5））

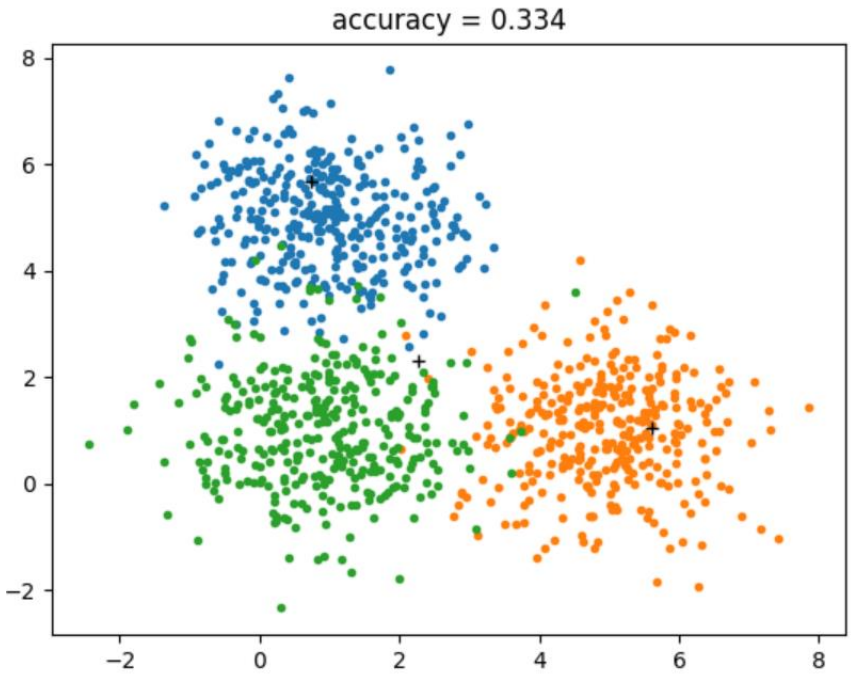


准确率 68.7%（明显感觉随着类的增加，样本更接近，效果更差）
得到的中心（其实中心的偏差不大，就是正确率存在贝叶斯 error）

```
[[3.25194016 0.978803  ]
 [0.68213428 3.24591784]
 [2.89114085 3.30412714]
 [0.91470141 0.93231929]]
```

4.2 GMM-EM

因为 GMM-EM 对初值非常的敏感，所以我们将 K-means 得到的结果作为 GMM 的初值，这样就不会有太大的偏离。
（三个类，中心分别为（1，5） （5，1） （1，1））



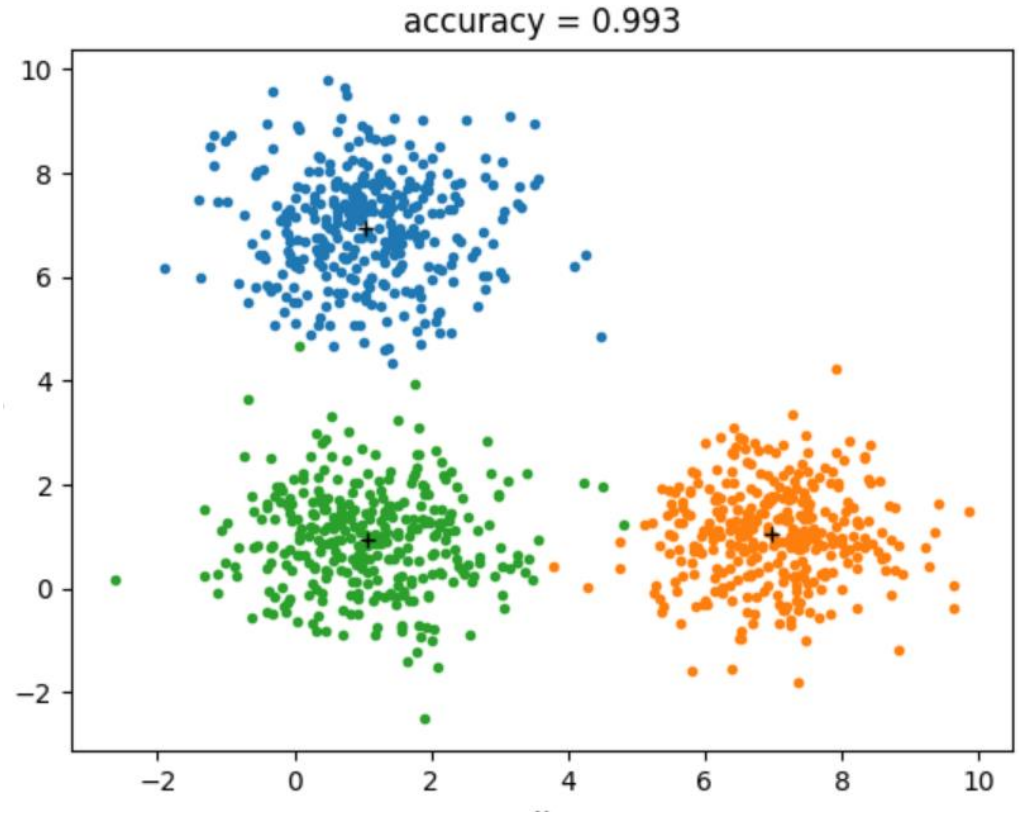
准确率 33.4%，（分类效果并不好，所以我们打算将样本分开一点）
得到的中心、α、协方差矩阵：

```
[[0.72587846 5.67975633]
 [2.27273041 2.31411757]
 [5.6056628  1.04158665]]
```

```
[[7.68384928e-07]
 [9.99996594e-01]
 [2.63782266e-06]]
```

```
[[ 0.23102981 -0.04031976]
 [-0.04031976  0.17480985]]
[[ 4.58737242 -1.7622173 ]
 [-1.7622173   4.57214559]]
[[0.20272478 0.03499529]
 [0.03499529 0.26297861]]
```

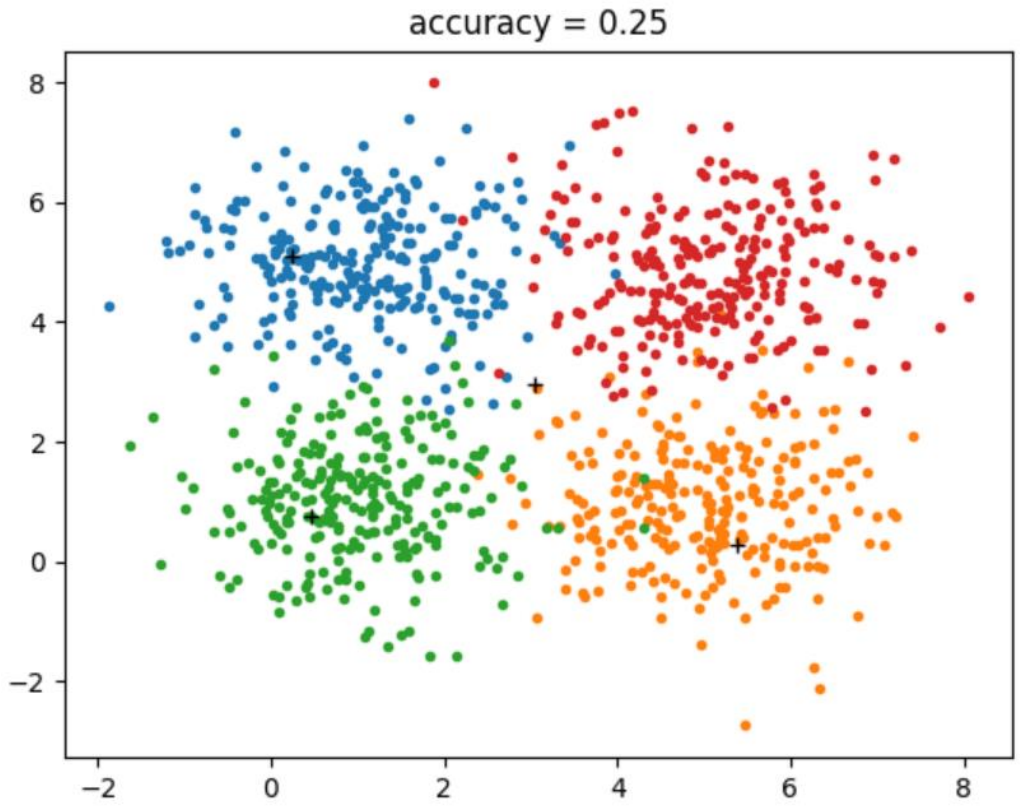
（三个类，中心分别为（1，7） （7，1） （1，1））



准确率：99.3%（将样本适当分开效果还是不错的。）
得到的中心、α、协方差矩阵：

```
中心：
[[7.0467065  0.95248063]
 [0.99061046 6.88532139]
 [0.97293436 1.07368631]]
alpha：
[[0.33299794]
 [0.33438604]
 [0.33261602]]
协方差矩阵：
[[ 0.98934505 -0.00806698]
 [-0.00806698  0.95596598]]
[[0.97305103 0.03218902]
 [0.03218902 0.9788601 ]]
[[0.98034976 0.0012783 ]
 [0.0012783  1.03387154]]
```

（四个类，中心分别为（1，5） （5，1） （1，1） （5，5））

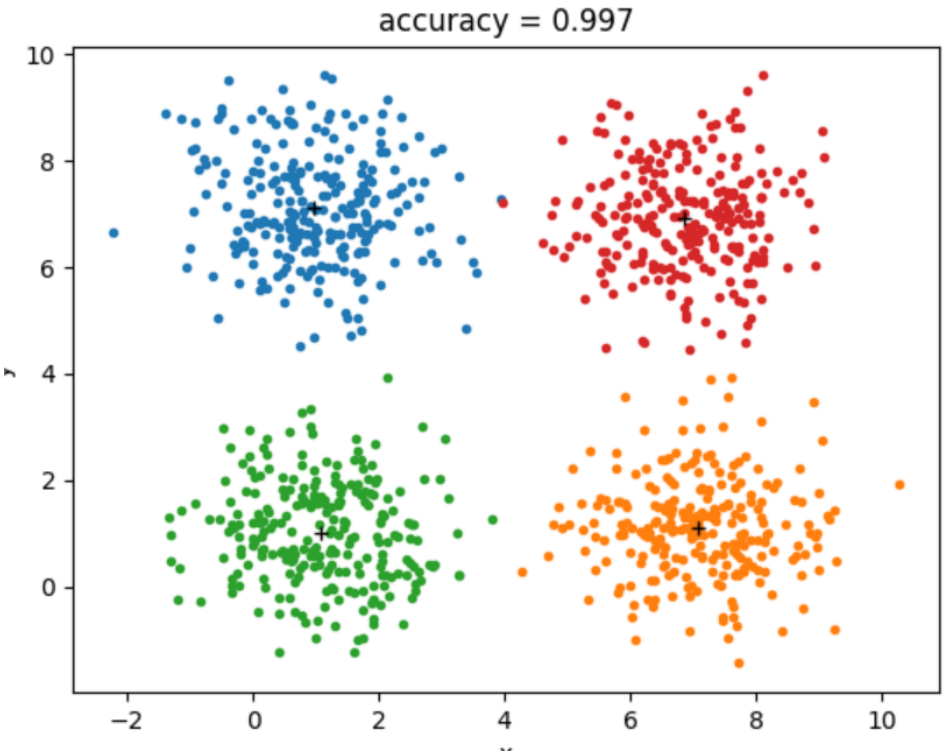


准确率 25%（怎么说呢，效果还是让人堪忧啊）
得到的中心、α、协方差矩阵：

```
中心：
[[0.24750939 5.10845514]
 [5.38200864 0.28796631]
 [0.46157432 0.77298452]
 [3.03653636 2.95653344]]
alpha：
[[2.28482626e-30]
 [1.69280393e-26]
 [1.79662003e-25]
 [1.00000000e+00]]
协方差矩阵：
[[ 0.02741196 -0.00968957]
 [-0.00968957  0.01366269]]
[[ 0.10095858 -0.02637133]
 [-0.02637133  0.02092997]]
[[ 0.06272857 -0.01233526]
 [-0.01233526  0.04445364]]
[[ 5.00779382 -0.03584304]
 [-0.03584304  4.86293338]]
```

所以我们继续适当将 4 个类分开一点

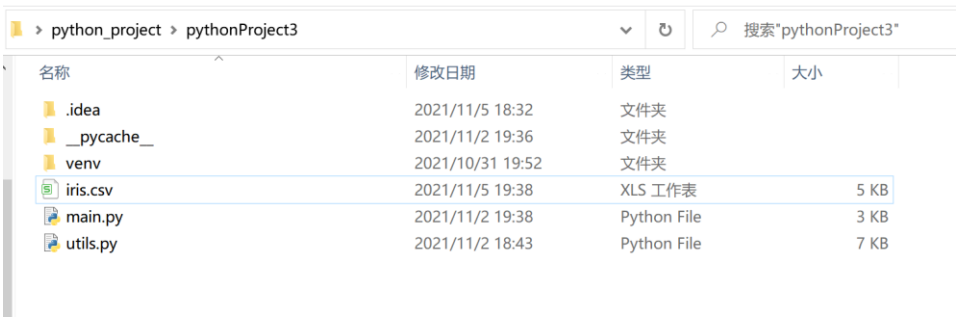
（四个类，中心分别为（1，7） （7，1） （1，1） （7，7））



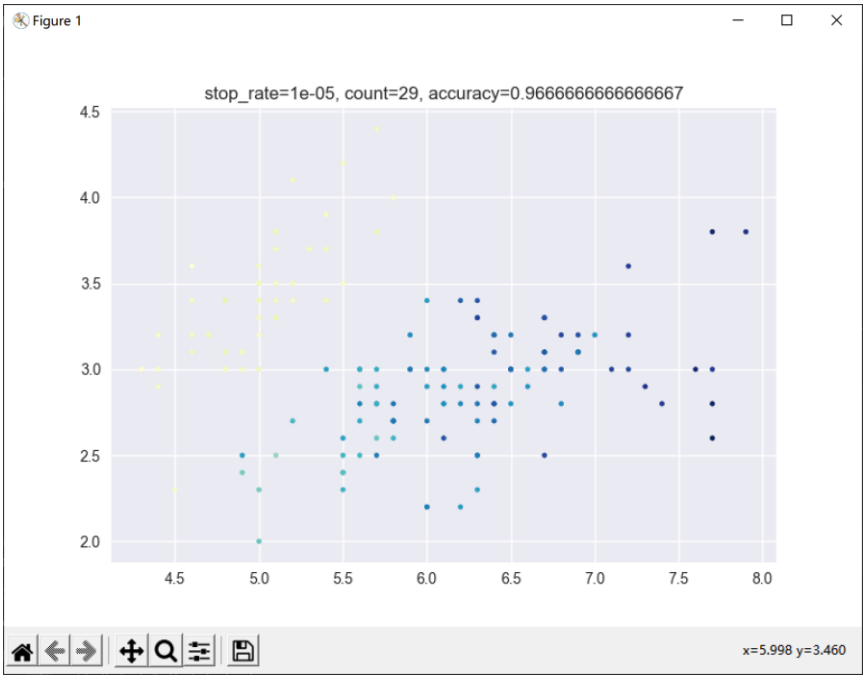
准确率 99.7%（效果很好）
得到的中心、α、协方差矩阵：

```
中心：
[[1.08996856 1.01912619]
 [7.09433766 1.12152269]
 [6.85759176 6.91472136]
 [0.98342771 7.1297552 ]]
alpha：
[[0.24996392]
 [0.24925578]
 [0.25103924]
 [0.24974106]]
协方差矩阵：
[[ 1.00588869 -0.07757422]
 [-0.07757422  0.9162156 ]]
[[ 1.02341837 -0.02194536]
 [-0.02194536  0.84340638]]
[[0.89945595 0.011644  ]
 [0.011644   1.00177927]]
[[ 0.99344913 -0.10613754]
 [-0.10613754  1.08474899]]
```

4.3 UCI 数据



sepal_len	sepal_wid	petal_len	petal_wid	class
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa
4.8	3.4	1.6	0.2	Iris-setosa
4.8	3	1.4	0.1	Iris-setosa
4.3	3	1.1	0.1	Iris-setosa
5.8	4	1.2	0.2	Iris-setosa
5.7	4.4	1.5	0.4	Iris-setosa
5.4	3.9	1.3	0.4	Iris-setosa
5.1	3.5	1.4	0.3	Iris-setosa
5.7	3.8	1.7	0.3	Iris-setosa
5.1	3.8	1.5	0.3	Iris-setosa
5.4	3.4	1.7	0.2	Iris-setosa
5.1	3.7	1.5	0.4	Iris-setosa
4.6	3.6	1	0.2	Iris-setosa
5.1	3.3	1.7	0.5	Iris-setosa
4.8	3.4	1.9	0.2	Iris-setosa
5	3	1.6	0.2	Iris-setosa
5	3.4	1.6	0.4	Iris-setosa
5.2	3.5	1.5	0.2	Iris-setosa
5.2	3.4	1.4	0.2	Iris-setosa
4.7	3.2	1.6	0.2	Iris-setosa
4.8	3.1	1.6	0.2	Iris-setosa
5.4	3.4	1.5	0.4	Iris-setosa
5.2	4.1	1.5	0.1	Iris-setosa



五、结论

1. K-means和GMM都是EM算法的体现。两者共同之处都有隐变量，遵循EM算法的E步和M步的迭代优化。不同之处在于K-means给出了很多很强的假设，比如假设了所有聚类模型对总的贡献是相等的（平均的），假设一个样本由某一个特定聚类模型产生的概率是1，其他为0. 而GMM用混合高斯模型来描述聚类结果。假设多个高斯模型对总模型的贡献是有权重的，且样本属于某一类也是
2. 由概率的。两者都能较好的解决简单的分类问题，但存在着可能只取到局部最优的问题。初值的选取对K-means和GMM的效果影响较大。K-means的初值选取通常是给定聚类个数k和随机选取初始聚类中心。而对于GMM来说，如果初始高斯模型的均值和方差选取不好的话，可能会出现极大似然值为0的情况，即该样本几乎不可能由我们初始的高斯模型生成。另外在实验过程中还会出现协方差矩阵不可逆的情况
3. EM算法对初值的敏感程度更高，可以用K-Means得到的结果作为EM算法的初值；
4. 根据K-Means算法的原理可知，其主要依赖于对欧式距离的求取，所以实验的优缺点都可以在距离取得不同中展现出来；
5. 其实K-Means就是一种特殊的高斯混合模型，假设每一类在样本数据中出现的概率相等，即均为1/k 。而且假设高斯模型中的每个变量之间是独立的，即变量间的协方差矩阵是对角阵，这样我们可以直接用欧氏距离作为K-Means的协方差去衡量相似性。
6. 对比不同分类数下K_Means与EM方法的分类效果，K-Means的分类效果都好于EM方法，而且前者的迭代次数比后者少

六、参考文献

[1]机器学习/周志华著.—北京：清华大学出版社，2016 ISBN 978-7-302-42328-7

[2]统计学习方法/李航著.-2 版.-北京：清华大学出版社，2019（2020.11 重印）ISBN 978-7-302-51727-6

七、附录：源代码（带注释）

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


def get_uci():
    path = "Exasens.csv"
    data_set = pd.read_csv(path)
    x = data_set['Diagnosis']
    y = data_set.drop('Diagnosis', axis=1)
    _label, _data = np.array(x, dtype=str), np.array(y, dtype=int)
    for i in range(_label.shape[0]):
```

```
            if _label[i] == "COPD":
                _label[i] = 0
            elif _label[i] == "HC":
                _label[i] = 1
            elif _label[i] == "Asthma":
                _label[i] = 2
            elif _label[i] == "Infected":
                _label[i] = 3
        labels = np.array(_label, dtype=int)
        print()
        return labels, _data


def generate_data(k, count):
        means = np.zeros((k, 2))                            # 各个高斯分布的均值
        _data = np.zeros((count, 3))                        # 待分类的数据
        for i in range(k):
            means[i, 0] = float(input("输入第 " + str(i + 1) + " 组数据的均值\n"))
            means[i, 1] = float(input())
        cov = np.array([[1, 0], [0, 1]])                    # 协方差矩阵
        index = 0
        for i in range(k):
            for j in range(int(count / k)):
                _data[index, 0:2] = np.random.multivariate_normal(means[i], cov)
                _data[index, 2] = i
                index += 1
        return means, _data


# 找到最小距离
def get_min(distance):
        value = distance[0, 0]
        index = 0
        for i in range(1, distance.shape[1]):
            if value > distance[0, i]:
                value = distance[0, i]
                index = i
        return index


# k-means 算法
def k_means(sample, num):
        cnt= -1                                             # 记录迭代次数
        dimension = sample.shape[1]                         # 数据的维度
```

```
    all_list = []                                          # 存放 k 个簇
    center = np.zeros((num, dimension))                    # 记录 k 个簇的中心
    maxi = len(sample)
    rand = np.random.randint(0, maxi, num)                 # 随机选取 k 个样本初始化
    for i in range(num):
        temp_list = [rand[i]]
        all_list.append(temp_list)
    while True:                                            # 迭代
        cnt += 1
        old_center = center.copy()                         # 记录更新之前的中心
        for i in range(num):
            cluster = all_list[i]
            length = len(cluster)
            sums = np.zeros((1, dimension))                # 记录一个簇所有数据各维度的加和
            for j in range(dimension):                     # 计算每个簇的中心
                for f in range(length):
                    sums[0, j] += sample[cluster[f], j]
                center[i, j] = float(sums[0, j] / length)  # 得到中心的各个维度
        if np.sum(abs(center - old_center)) < 0.1:         # center 基本不变，结束迭代
            print(cnt)
            return center, all_list
        for i in range(num):
            all_list[i].clear()
        index = -1
        for info in sample:
            index += 1
            distance = np.zeros((1, num))                  # 记录一个数据到三个中心的距离
            for i in range(num):
                for j in range(dimension):
                    distance[0, i] += float((center[i, j] - info[j]) ** 2)
            category = get_min(distance)                   # 找到距离该数据最近的中心
            all_list[category].append(index)               # 划分簇


# 初始化先验概率、均值和协方差矩阵
def init(center, sample, num):
    dimension = sample.shape[1]
    # u = np.zeros((num, dimension))                       # 初始化均值
    u = center.copy()
    covariance = np.eye(dimension)                         # 初始化协方差
    cov = []
    alpha = np.zeros((num, 1))                             # 初始化先验
    # maxi = len(sample)
    # rand = np.random.randint(0, maxi, num)               # 随机选取 k 个样本初始化
```

```
    for i in range(num):
        # u[i, :] = sample[rand[i]]
        alpha[i, 0] = float(1 / num)
        cov.append(covariance)
    return u, alpha, cov


def gaussian_probability(x, u, covariane, num):
    delta = x.reshape(len(x), 1) - u.reshape(len(u), 1)
    covariane1 = pow(np.linalg.det(covariane), 0.5)
    index1 = -num / 2
    pai = pow((2 * np.pi), index1)
    index2 = (-1/2) * np.dot(delta.T, np.linalg.inv(covariane)).dot(delta)
    prior = (covariane1 * pai * np.exp(index2))
    return prior


# 通过最大似然得到均值、协方差矩阵和先验概率
def like_hood(sample, gama, num):
    dimension = sample.shape[1]
    number = sample.shape[0]
    u = np.zeros((num, dimension))          # 更新均值
    cov = []                                  # 更新协方差矩阵
    alpha = np.zeros((num, 1))              # 更新先验
    # 更新均值
    for i in range(num):
        for j in range(dimension):
            divisor = 0
            dividend = 0
            for f in range(number):
                divisor += gama[f, i]
                dividend += gama[f, i] * sample[f, j]
            u[i, j] = float(dividend / divisor)
    # 更新协方差矩阵
    for i in range(num):
        divisor = 0
        dividend = np.zeros((dimension, dimension))
        for j in range(number):
            delta = sample[j].reshape(dimension, 1) - u[i, :].reshape(dimension, 1)
            matrix = np.dot(delta, delta.T)
            dividend += gama[j, i] * matrix
            divisor += gama[j, i]
        covariance = dividend / divisor
        cov.append(covariance)
```

```
    # 更新先验概率
    for i in range(num):
        gama_sum = 0
        for j in range(number):
            gama_sum += gama[j, i]
        alpha[i, 0] = float(gama_sum / number)
    return u, cov, alpha


# 通过均值、协方差矩阵和先验概率求得后验概率
def gaussian_mixture(sample, num, u, alpha, cov):
    number = sample.shape[0]
    gama = np.zeros((number, num))
    total_probability = []                                  # 全概率
    for j in range(number):
        sums = 0
        for i in range(num):
            sums += float(alpha[i, 0] * gaussian_probability(sample[j, :], u[i, :], cov[i], num))
        total_probability.append(sums)                      # 计算全概率

    for j in range(number):
        for i in range(num):                                # 计算 xj 属于 i 类的后验概率
            gama[j, i] = float(alpha[i, 0] * gaussian_probability(sample[j, :], u[i, :], cov[i],
num)
                               / total_probability[j])
    return gama


# EM 算法
def em_algorithm(center, sample, k):
    u, alpha, cov = init(center, sample, k)    # 初始化参数
    iterator = 0                               # 记录迭代次数
    while True:
        iterator += 1                          # 迭代次数
        prev_u = u.copy()                      # 记录上一次迭代的参数
        # prev_alpha = alpha.copy()
        # prev_cov = cov.copy()
        gama = gaussian_mixture(sample, k, u, alpha, cov)       # E 步

        u, cov, alpha = like_hood(sample, gama, k)                      # M 步
        if np.sum(abs(prev_u - u)) < 0.05:  # 均值基本不变，结束迭代
            print(iterator - 1)
            break
    cluster = classify(sample, k, u, cov, alpha)
```

```
    return u, cov, alpha, cluster


# 划分簇
def classify(sample, num, u, cov, alpha):
    all_list = []
    for i in range(num):
        temp = []
        all_list.append(temp)                          # 创建 num 个簇
    gama = gaussian_mixture(sample, num, u, alpha, cov)
    for i in range(gama.shape[0]):
        index = 0
        maxi = gama[i, 0]
        for j in range(1, gama.shape[1]):
            if gama[i, j] > maxi:
                maxi = gama[i, j]
                index = j                              # 寻找最大的 gama 来划分簇
        all_list[index].append(i)
    return all_list


# 获得最大数的索引
def get_attribute(label_set):
    maxi = label_set[0, 0]
    for i in range(1, label_set.shape[0]):
        if maxi < label_set[i, 0]:
            maxi = label_set[i, 0]
    return maxi


# 准确率分析
def accuracy(cluster, labels, num):
    sum_classified = 0
    for i in range(k):
        label_set = np.zeros((num, 1))
        for index in cluster[i]:
            index1 = int(labels[index])
            label_set[index1, 0] += 1
        sum_classified += get_attribute(label_set)
    my_accuracy = float(sum_classified / labels.shape[0])
    return my_accuracy


# 绘图
```

```
def draw_point(point_set, num, center, accurate):
    count = int(len(point_set) / num)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("accuracy = " + str(accurate))
    for i in range(num):
        up = (i + 1) * count
        down = i * count
        set_x = point_set[down: up, 0]
        set_y = point_set[down: up, 1]
        plt.plot(set_x, set_y, linestyle='', marker='.')
    for i in range(num):
        plt.plot(center[i, 0], center[i, 1], linestyle='', marker='+', color='Black')
    plt.show()


if __name__ == '__main__':
    k = 4                           # 高斯分布个数
    data_num = 1000                 # 待分类的数据集大小


    # label, data = get_uci()
    # center1, cluster1 = k_means(data, data.shape[1])
    # center2, covariances, alphas, cluster2 = em_algorithm(center1, data, data.shape[0])


    mean1, data_label = generate_data(k, data_num)

    label1 = data_label[:, 2].reshape(1, data_num)
    label = np.array(label1[0], dtype=int)

    data = data_label[:, 0:2].reshape(data_num, 2)

    centers1, clusters1 = k_means(data, k)
    print(mean1)
    print(centers1)

    # centers2, covariances, alphas, clusters2 = em_algorithm(centers1, data, k)
    # print(centers2)
    # print(alphas)
    # for covs in covariances:
    #     print(covs)

    draw_point(data, k, centers1, accuracy(clusters1, label, k))
```

```
# draw_point(data, k, centers2, accuracy(clusters2, label, k))
```