哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2020 年春季学期

计算学部《机器学习》课程

Lab 3 实验报告

姓名	王科龙
学号	1180801203
班号	1803104
电子邮件	1264405807@qq.com
手机号码	19917620613

目录

1 问题描述

1.1

实现一个 k-means 算法和混合高斯模型，并且使用 EM 算法估计模型的参数。

测试：

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）。

（1）用 k-means 聚类，测试效果；

（2）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）。

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 算法原理

2.1 k-mean 算法

k-mean 算法的的伪代码是：

1. 选择 k 个样本点，作为每一类的中心点
2. 计算每一个样本点距离中心点的距离，将其划分到距离最近的中心点那一类
3. do
4. 重新计算每一类的中心点，公式为$\overrightarrow{\mu_k} = \frac{1}{|C_k|}\sum_{i \in C_k} \overrightarrow{x_i}$, C_k表示第 k 类
5. 重新计算每一个样本点距离中心点的距离，将其划分到距离最近的中心点那一类
6. until(中心点的变化小于精度)

   K-mean 算法的结果和初值和很大的关系，不同的初值，最后的结构哦可能不同因此可以通过选择较好的初值来运行 k-mean，来得到较好的结果

2.2 GMM 模型

GMM 模型为

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{z}} p(\boldsymbol{z}) p(\boldsymbol{x}|\boldsymbol{z}) = \sum_{k=1}^{K} \pi_k N(\mu_k, \Sigma_k)$$

其中$\boldsymbol{z}$为二值隐变量，$p(z_k = 1) = \pi_k$，表示该数据取自第 k 个高斯分布的概率。

$N(\mu_k, \Sigma_k)$表示均值为μ_k,协方差矩阵为Σ_k的正态分布，表达式如下：

$$p(x|\mu_k, \Sigma_k) = \frac{1}{(2\pi)^{\frac{D}{2}} |\Sigma_k|^{\frac{1}{2}}} \exp\left(-\frac{(x_n - \mu_k)^T \Sigma_k^{-1} (x_n - \mu_k)}{2}\right)$$

某样本属于某一类的后验概率：以下公式表示样本x属于类别k的概率

$$\gamma(z_k) = p(z_k = 1|x) = \frac{p(z_k = 1) p(x|z_k = 1)}{\sum_{j=1}^{K} \pi_k N(x|\mu_j, \Sigma_j)} = \frac{\pi_k * N(x|\mu_k, \Sigma_k)}{p(x)}$$

通过算出某一个样本属于第 k 类的概率，比较属于其他类的概率，选择使得这个概率最大的 k 做为这个样本的分类。这样就完成了聚类任务

为了计算这个后验概率，需要估计 GMM 模型的参数π_k, μ_k, Σ_k。

采用 MLE 进行估计，那么对于给定样本集D，对数似然函数是

$$LLD(D) = \sum_{n=1}^{N} \ln\left(\sum_{k=1}^{K} \pi_k N(\boldsymbol{x}|\boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})\right)$$

分别对三个参数求导，令其导数等于 0，可得：

其中第三个参数有限制条件$\sum_{k=1}^{K} \pi_k = 1$,计算最值时需要使用拉格朗日乘数法。

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k}\sum_n \gamma(z_{nk})\boldsymbol{x_n}, \quad N_k = \sum_n \gamma(z_{nk})$$

$$\boldsymbol{\Sigma_k^{new}} = \frac{\mathbf{1}}{N_k}\sum_{\boldsymbol{n=1}}^{\boldsymbol{N}} \gamma(z_{nk})(\boldsymbol{x_n^{new}} - \boldsymbol{\mu_k})(\boldsymbol{x_n^{new}} - \boldsymbol{\mu_k})^T$$

$$\pi_k^{new} = \frac{N_k}{N}$$

但是这个结果是迭代式，无法直接求解出参数。因此采用 EM 算法来计算参数。

E-Step：

对于每一个样本$\boldsymbol{x_n}$，利用π_k, μ_k, Σ_k计算其对于每一类的后验概率$\gamma(z_{nk})$

形成γ矩阵

M-Step:

通过公式进行参数更新

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k}\sum_n \gamma(z_{nk})\boldsymbol{x_n}, \quad N_k = \sum_n \gamma(z_{nk})$$

$$\boldsymbol{\Sigma_k^{new}} = \frac{\mathbf{1}}{N_k}\sum_{\boldsymbol{n=1}}^{\boldsymbol{N}} \gamma(z_{nk})(\boldsymbol{x_n^{new}} - \boldsymbol{\mu_k})(\boldsymbol{x_n^{new}} - \boldsymbol{\mu_k})^T$$

$$\pi_k^{new} = \frac{N_k}{N}$$

伪代码:

1. 初始化参数$\pi_k, \mu_k, \Sigma_k \ \ k = 1,2, \dots, K$
2. $E - step$
3. **DO**
4. $for\ k = 1,2,3 \dots, K$:
5. $\quad M - step(k)$
6. $\quad$ 更新参数π_k, μ_k, Σ_k
7. $E - step$
8. **UNTIL：**满足停止条件
9. $for\ i = 1,2, \dots, N$:
10. $\quad$ 将样本$\boldsymbol{x}_n$划分到后验概率最大的那一类中

3 实验做法

3.1 生成数据

通过自定均值和协方差，采用 np.random.multivariate_normal 生成高维高斯分布数据

3.2 K-mean

依照伪代码的逻辑进行计算

```
def k_mean(self):
    # 选择k个点当作起始的mean,get label
    for i in range(self.k):
        self.data[self.n, i] = i  # 标签在self.n这一行上
    self.mean = self.data[:, :self.k]
    old_mean = None
    time = 0
    while True:
        self.get_label()  # 贴标签
        # 重新计算mean
        old_mean = self.mean
        self.mean = self.get_mean()
        print(self.mean)
        print()
        if self.get_mean_diff(old_mean) <= self.precision: break
        time = time + 1
        if time > self.times:
            break
```

data 为 data_size * (n+1)大小的矩阵，前 n 行是各个维度，最后一行是各个样本的标签
循环跳出条件通过迭代次数和前后各个中心点二范数之和的变化小于精度来控制
为所有数据贴标签的做法：

```
def get_label(self):
    labels = [0] * self.data_size
    for i in range(self.data_size):  # 计算每一个点的标签
        distance = float('inf')
        point = self.data[:self.n, i]  # 取第i列，并且不要最后一行标签
        for j in range(self.k):  # 计算该点距离哪个center点最近
            center = self.mean[:self.n, j]  # 取mean中的第j类的中心点
            if self.get_distance(center, point) < 1e-6:
                distance = 0
                labels[i] = self.mean[self.n, j]
            elif self.get_distance(center, point) < distance:
                labels[i] = self.mean[self.n, j]
                distance = self.get_distance(center, point)
    self.data[self.n, :] = np.mat(labels)
```

距离的计算采用欧氏距离的计算方法：

```
def get_distance(self, point, center):
    return float((center - point).T * (center - point))
```

重新计算中心点：

```
# 根据已有mean重新计算mean
def get_mean(self):
    ret = np.mat(np.zeros((self.n, 1)))  # 代表mean,没有加入标签中的一列
    for i in range(self.k):  # 计算第k类的均值(center)
        temp = np.mat(np.zeros((self.n, 1)))  # 第k类的暂时均值，没有加入标签
        count = 0
        for j in range(self.data_size):  # 看第j个点
            if self.data[self.n, j] != self.mean[self.n, i]:  # 标签不同跳过
                continue
            else:
                temp = temp + self.data[:self.n, j]
                count = count + 1
        temp = 1 / count * temp
        if ret.shape == temp.shape and float(ret.T * ret) == 0:
            ret = temp
        else:
            ret = np.hstack([ret, temp])
    # 增加每个中心点的标签
    ret = np.vstack([ret, self.mean[self.n, :]])
    return ret
```

3.3 GMM

同样依照伪代码的逻辑写代码：

```python
def parse_param(self):
    # 参数在初始化在init中完成
    self.e_step()
    old_value = 0
    new_value = self.log_likelihood()
    time = 0
    while True:
        for i in range(self.k):  # 计算每一类的参数
            self.m_step(i)
        self.e_step()
        time = time + 1
        print(time)
        print(self.gamma)
        print()
        old_value = new_value
        new_value = self.log_likelihood()
        if np.abs(new_value - old_value) < self.precision: break
        if time > self.times: break
    self.get_label()
```

循环的跳出条件通过循环次数和似然函数值的变化小于精度来控制

初始化过程:

```python
def __init__(self, k, data, precision, times):
    self.k = k
    self.data_size = data.shape[1]  # 分类数据的个数
    self.d = data.shape[0]  # 特征数据的维度
    self.data = data
    self.label = [0] * self.data_size  # 每个数据的标签
    self.precision = precision
    self.times = times
    # 初值
    self.mu = [np.mat(np.zeros((self.d, 1)))] * self.k  # 有k个mu，每个mu都是d维的
    for _k in range(self.k):
        self.mu[_k] = self.mu[_k] + self.data[:, _k]
    self.pi = [1 / k] * self.k  # 有k个pi,每个pi都是数字
    self.sigma = [np.mat(np.eye(self.d))] * self.k  # 有k个sigma,每个sigma都是dxd的矩阵
    self.gamma = np.mat(np.zeros((self.k, self.data_size)))  # 包括所有r(znk)的矩阵
```

$E-step$:

```python
def e_step(self):  # 计算gamma的每一项
    for i in range(self.data_size):  # 对每一个样本
        for j in range(self.k):  # 对每一个类别
            self.gamma[j, i] = self.get_posterior(j, i)
```

其中涉及计算后验概率，带入公式进行计算:

```
def get_posterior(self, k, data_num):  # 得到某个样本对于分类k的后验
    numerator = self.pi[k] * self.get_gauss(self.mu[k], self.sigma[k], self.data[:, data_num])
    denominator = self.get_px(data_num)
    return numerator / denominator
```

计算高斯分布的概率密度，带入公式进行计算：

```
def get_gauss(self, mu, sigma, x):
    part1 = 1 / np.power(2 * np.pi, self.d / 2)
    part2 = 1 / np.power(np.linalg.det(sigma), 0.5)
    part3 = np.exp(-0.5 * (x - mu).T * sigma * np.linalg.inv(sigma) * (x - mu))
    return part1 * part2 * part3
```

计算$p(x)$,同样带入公式进行计算

```
def get_px(self, data_num):  # 贝叶斯公式的分母，同时也是x_n的概率计算公式
    result = 0
    for _k in range(self.k):
        result = result + self.pi[_k] * self.get_gauss(self.mu[_k], self.sigma[_k], self.data[:, data_num])
    return result
```

$M-step$

```
def m_step(self, k):  # 根据e_step的计算结果更新参数,更新了第k类的参数
    Nk = self.get_Nk(k)
    # 计算mu[k]
    mu_temp = np.mat(np.zeros((self.mu[k].shape[0], self.mu[k].shape[1])))
    for data_num in range(self.data_size):
        mu_temp = mu_temp + self.gamma[k, data_num] * self.data[:, data_num]
    mu_temp = 1 / Nk * mu_temp
    # 计算 sigma[_k]
    sigma_temp = np.mat(np.zeros((self.sigma[k].shape[0], self.sigma[k].shape[1])))
    for data_num in range(self.data_size):
        sigma_temp = sigma_temp + self.gamma[k, data_num] * (self.data[:, data_num] - mu_temp) * (
                self.data[:, data_num] - mu_temp).T
    sigma_temp = 1 / Nk * sigma_temp
    # 计算pi[_k]
    pi_temp = Nk / self.data_size
    # 更新
    self.mu[k] = mu_temp
    self.sigma[k] = sigma_temp
    self.pi[k] = pi_temp
```

N_k的计算结果是 Nk，带入公式计算：

```
def get_Nk(self, k):  # 所有样本对于分类k的后验的累加和
    sum = 0
    for i in range(self.data_size):  # 第k行所有项求和
        sum = sum + self.gamma[k, i]
    return sum
```

在$M-step$中计算π_k, μ_k, Σ_k同样带入公式计算，然后进行更新

将参数计算完成后，需要根据每个样本对于每一类的后验概率贴标签确定分类，这些后验概率都保存在γ矩阵中

```
def get_label(self):
    for data_num in range(self.data_size):  # 计算每一个元素的分类
        max = 0
        for _k in range(self.k):
            if self.gamma[_k, data_num] > self.gamma[max, data_num]:
                max = _k
        self.label[data_num] = max
```

3.4 在 UCI 数据集上测试

数据来自 http://archive.ics.uci.edu/ml/datasets/DrivFace
数据介绍：
filename：图片的名字
subject：图片来自于的司机编号可取 1~4
imgNum：图片编号
label：图片标签，可取 1，2，3 分别表示 3 中不同的注视方向，本次测试只采集第 2 类数据
ang：注释的角度
xF,yF,wF,hF：Face position
xRE,yRE：right eye position
xLE,yLE：left eye position
xN,yN：Nose position
xRM,yRM：right corner of mouth
xLM,yLM：left corner of mouth
本次测试只采集第二类数据，然后将这些数据归为 4 类，分属于不同的司机
采集数据之后调用相应的方法计算即可。
但是由于数据数值过大，在计算 GMM 时指数会溢出，因此在 GMM 计算前，对数据进行了归一化处理，同时为了扩大差距，又人为将数值放大了 5 倍

```python
def data_normal(data):
    row = data.shape[0]
    column = data.shape[1]
    ret = []
    for i in range(row):
        max = data[i, 0]
        min = data[i, 0]
        for j in range(column):
            if max < data[i, j]:
                max = data[i, j]
            if min > data[i, j]:
                min = data[i, j]
        for j in range(column):
            a = 5 * (data[i, j] - min) / (max - min)
            # data[i, j] = (data[i, j] - min) / (max - min)
            ret.append(a)
    ret = np.mat(np.array(ret).reshape((row, column)))
    return ret
```

聚类效果的评价指标来自于西瓜书上的外部指标:

对于数据$D=\{x_1,x_2,x_3\ldots,x_m\}$,假定通过聚类给出的簇划分为$C=\{C_1,C_2,\ldots,C_k\}$,参考模型给出的簇划分为$C^*=\{C_1^*,C_2^*,\ldots C_k^*\}$,令$\boldsymbol{\lambda}$和$\boldsymbol{\lambda}^*$分别表示$C$和$C^*$对应的簇标记向量，将样本两两配对考虑，定义

$$a=|SS|,SS=\{(\boldsymbol{x_i},\boldsymbol{x_j})|\boldsymbol{\lambda}_i=\boldsymbol{\lambda}_j,\boldsymbol{\lambda}_i^*=\boldsymbol{\lambda}_j^*,i<j\}$$
$$b=|SD|,SD=\{(\boldsymbol{x_i},\boldsymbol{x_j})|\boldsymbol{\lambda}_i=\boldsymbol{\lambda}_j,\boldsymbol{\lambda}_i^*\neq\boldsymbol{\lambda}_j^*,i<j\}$$
$$c=|DS|,DS=\{(\boldsymbol{x_i},\boldsymbol{x_j})|\boldsymbol{\lambda}_i\neq\boldsymbol{\lambda}_j,\boldsymbol{\lambda}_i^*=\boldsymbol{\lambda}_j^*,i<j\}$$
$$a=|DD|,DD=\{(\boldsymbol{x_i},\boldsymbol{x_j})|\boldsymbol{\lambda}_i\neq\boldsymbol{\lambda}_j,\boldsymbol{\lambda}_i^*\neq\boldsymbol{\lambda}_j^*,i<j\}$$

则有:

Jaccard 系数

$$JC=\frac{a}{a+b+c}$$

FM 指数:

$$FM=\sqrt{\frac{a}{a+b}\cdot\frac{a}{a+c}}$$

Rand 指数:

$$RI=\frac{2(a+d)}{m(m-1)}$$

在本次实验中我采用 Rand 指数作为评价指标:

```python
def get_rate(ori_label, rate_label):  # 准确率评估
    a = 0
    b = 0
    c = 0
    d = 0
    data_size = ori_label.shape[1]
    for i in range(data_size):
        for j in range(data_size):
            if i < j:
                if rate_label[0, i] == rate_label[0, j]:
                    if ori_label[0, i] == ori_label[0, j]:
                        a = a + 1
                    else:
                        b = b + 1
                else:
                    if ori_label[0, i] == ori_label[0, j]:
                        c = c + 1
                    else:
                        d = d + 1
    return 2 * (a + d) / (data_size * (data_size - 1))
```

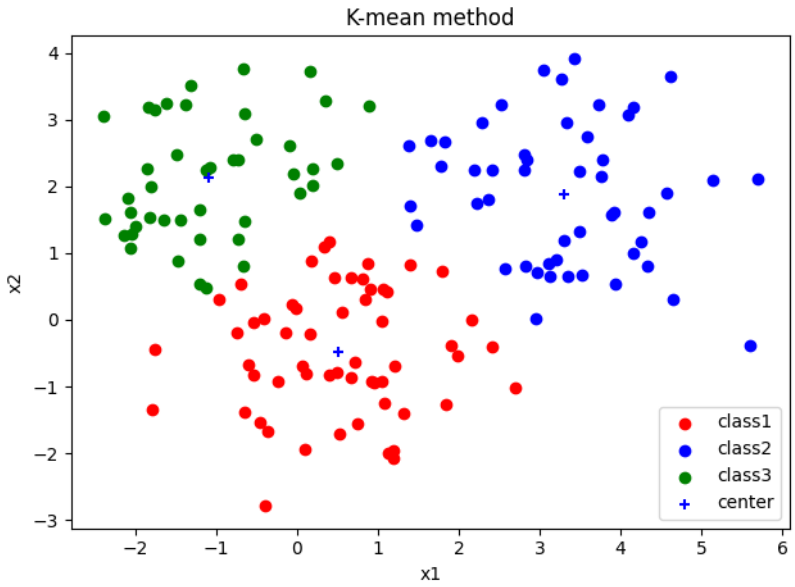
4 实验结果分析

4.1 k-mean 聚类结果

二维数据，设超参 k=3
人工生成的数据的均值为：

```
mu = [[-1, 2], [3, 1.5], [0.5, -0.8]]
train_data = generate_data(k, type_size, mu)
```

每一类有 50 个数据，迭代次数为 100 次
结果如下：



计算完成之后的均值为：

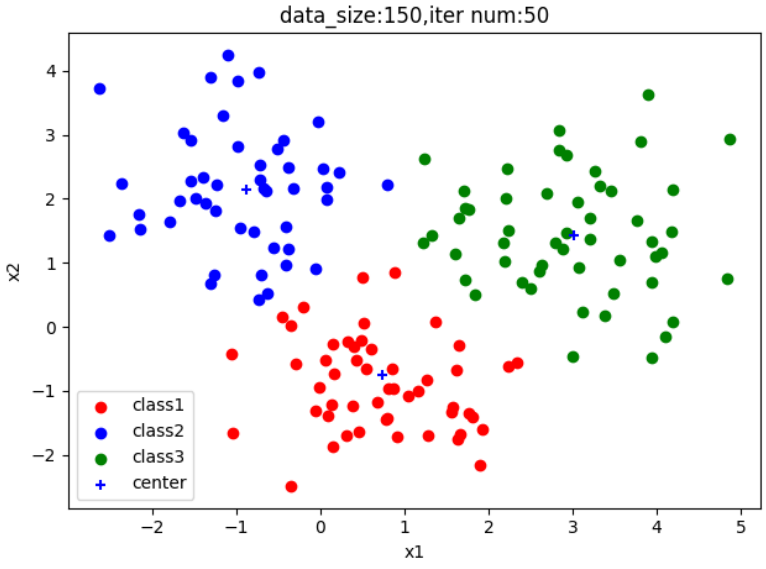
```
[[ 0.50873133  3.30240559 -1.08814653]
 [-0.48024768  1.88785946  2.1225656 ]
 [ 0.          1.          2.        ]]
```

4.2 GMM 模型

同样手工生成二维高斯分布数据，k=3，人工生成数据的均值为：

```
mu = [[-1, 2], [3, 1.5], [0.5, -0.8]]
```

每一类有 50 个数据，迭代次数为 50 次，结果如下：



计算完成之后各个高斯分布的均值为：

```
[matrix([[ 0.73902778],
        [-0.74699927]]), matrix([[-0.87216824],
        [ 2.150857  ]]), matrix([[3.00713883],
        [1.4323088 ]])]
```

和最初的设定值较接近。

4.3 在 UCI 数据上测试 GMM

k-mean 的测试结果：

```
kmean rank Index: 0.9230836441845616
```

GMM 的测试结果：

```
GMM rank Index= 0.917175790570286
```

5 结论

(1) 初值选取无论对 k 均值算法还是 GMM 算法的影响都很大，选择一个好的初值，可以极大的帮助聚类算法取得更好的效果

(2) k 均值算法和高斯混合聚类有共同之处，k 均值算法的贴标签相当于使用 GMM 计算 E 步，E 步计算了改样本是任何一个分类的概率，最大的那个分类就是该样本的标签
M 步在更新参数，类似于 k-mean 中的重新计算各类中心点。只是 GMM 中需要计算的参数比较多而已。

(3) 在 GMM 的初值选取问题中需要注意：（1）协方差矩阵必须可逆，（2）$\sum_{k=1}^{K}\pi_k = 1$,选择初值时仍要遵循这个条件。

(4) GMM 模型相对于 k 均值算法而言能得到更多的信息，因为 GMM 计算得出的是概率，有时如果遇到某个样本有 0.51 的概率是第 1 类的，有 0.49 的概率是其他类的这种情况，无论机器将它划分到那一类都是不太合理的，这个时候机器可以将其交给人类判别，而 k 均值就无法做到这一点了