哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称： 机器学习

课程类型： 选修

实验题目：k-means 聚类方法和混合高斯模型

学号：1190201303
姓名： 王艺丹

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）。
2. 用 k-means 聚类，测试效果；
3. 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）。
4. 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

Windows 10；Anaconda 4.8.4；python 3.7.4；jupyter notebook 6.0.1

三、 概念设计思想（主要算法及数据结构）

3.1 k-means：

k 近邻法是基本且简单的分类与回归方法，k 近邻法的基本做法是：对给定的训练实例点和输入实例点，首先确定输入实例点的 k 个最近邻训练实例点，然后利用这 k 个训练实例点的类的多数来预测输入实例点的类。

k 近邻模型对应于基于训练数据集对特征空间的一个划分。 k 近邻法中，当训练集、距离度量、 k 值及分类决策规则确定后，其结果唯一确定。

k 近邻法三要素：距离度量、 k 值的选择和分类决策规则。常用的距离度量是欧氏距离及更一般的 pL 距离。 k 值小时， k 近邻模型更复杂； k 值大时， k 近邻模型更简单。 k 值的选择反映了对近似误差与估计误差之间的权衡，通常由交叉验证选择最优的 k 。

给定训练样本$X = \{x_1, x_2, ..., x_m\}$和划分聚类数量$k$，给出一个簇划分$C = C_1, C_2, ..., C_k$，使得该划分的**平方误差$E$**最小：

$$E = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||_2^2$$

其中，$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x_i$，它是簇$C_i$的均值向量。 刻画了簇内样本围绕簇的均值向量的紧密程度，E越小，簇内样本的相似度越高。

显然优化求解E的最小值为一个NP完全问题，采取贪心的策略，具体算法迭代流程如下:

1. 首先随机/按最远距离(最可能不属于同一类)生成k个初始中心点；
2. 根据初始化的均值向量给出样本集 的一个划分，样本距离那个簇的均值向量距离最近，则将该样本划归到哪个簇；
3. 再根据这个划分来计算每个簇内真实的均值向量，如果真实的均值向量与假设的均值向量相同，则假设正确；否则，将真实的均值向量作为新的假设均值向量，回到步骤一继续迭代求解；
4. 当新的中心点与上一个中心点的距离变化小于给定精度时停止迭代

伪代码如下：

Algorithm 1 K Means

Input: $D=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_m\}$, 聚类簇数 k

Output: 簇划分 $\mathcal{C}=\{C_1,C_2,\cdots,C_k\}$

从 D 中随机选择 k 个样本作为初始均值向量 $\{\boldsymbol{\mu_1},\boldsymbol{\mu_2},\cdots,\boldsymbol{\mu_k}\}$
repeat
  令 $C_i=\varnothing\,(1\leq i\leq k)$
  for $j=1,2\cdots,m$ **do**
    计算样本 $\boldsymbol{x}_j$ 与各均值向量 $\boldsymbol{\mu_i}\,(1\leq i\leq k)$ 的距离：$d_{ji}=\|\boldsymbol{x}_j-\boldsymbol{x}_i\|_2$;
    根据距离最近的均值向量确定 $\boldsymbol{x}_j$ 的簇标记：$\lambda_j=\arg\min_{i\in\{1,2,\cdots,k\}}d_{ji}$;
    将样本 $\boldsymbol{x}_j$ 划入相应的簇:$C_{\lambda_j}=C_{\lambda_j}\cup\{\boldsymbol{x}_j\}$;
  end for
  for $i=1,2,\cdots,k$ **do**
    计算新均值向量：$\boldsymbol{\mu}_i'=\frac{1}{|C_i|}\sum_{\boldsymbol{x}\in C_i}\boldsymbol{x}$;
    if $\boldsymbol{\mu}_i'\neq\boldsymbol{\mu_i}$ **then**
      将当前均值向量 $\boldsymbol{\mu_i}$ 更新为 $\boldsymbol{\mu}_i'$
    else
      保持当前均值向量不变
    end if
  end for
until 当前均值向量均未更新

关键步骤具体实现如下：

```
def __k_means(self):
    count = 1
    while True:
        print("K-means", count)
        c = collections.defaultdict(list)
        for i in range(self.data_rows):
            dis_ij = [cal_dis(self.data[i], self.mu[j]) for j in range(self.k)]
            lambda_j = np.argmin(dis_ij)
            c[lambda_j].append(self.data[i].tolist())
            self.data_attribution[i] = lambda_j
        new_mu = np.array([np.mean(c[i], axis=0).tolist() for i in range(self.k)])
        loss = np.sum(cal_dis(self.mu[i], new_mu[i]) for i in range(self.k))
        if loss > self.eps:
            self.mu = new_mu
        else:
            break
        count += 1
        print(self.mu)
    return self.mu, c
```

3.2 GMM：

对于n维样本空间X中的随机变量x，若x服从高斯分布，其概率密度函数为：

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

其中μ是n维均值向量，Σ是$n*n$的协方差矩阵；易知，高斯分布完全由均值向量μ和协方差矩阵Σ这两个参数决定，因此我们也将概率密度函数记为$p(x|\mu,\Sigma)$；定义高斯混合分布如下：

$$p(x_i)=\sum_{j=1}^{K}\pi_j p\left(x_i|u_j,\Sigma_j\right)$$

该分部共由k个混合成分组成，每个混合成分对应一个高斯分布。其中 μ_i 与 Σ_i 是第 i 个高斯混合成分的参数，而$\alpha_i>0$为相应的“混合系数”，且 $\sum_{i=1}^{k}\alpha_i=1$

假设样本的生成过程由高斯混合分布给出：首先，根据$\alpha_1,\ldots,\alpha_i$定义的先验分布选择高斯混合成分，即 $p\left(z_j=i\right)=\alpha_i$，其中$\alpha_i$是选择第$i$个混合成分的概率；然后，根据被选择的混合成分的概率密度进行采样，从而生成相应的样本.根据贝叶斯定理，z_j的后验分布如下式所示：

$$\begin{aligned}p_{\mathcal{M}}\left(z_j=i|\boldsymbol{x}_j\right)&=\frac{P\left(z_j=i\right)\cdot p_{\mathcal{M}}\left(\boldsymbol{x}_j|z_j=i\right)}{p_{\mathcal{M}}\left(\boldsymbol{x}_j\right)}\\&=\frac{\alpha_i\cdot p\left(\boldsymbol{x}_j|\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i\right)}{\sum_{l=1}^{k}\alpha_l\cdot p\left(\boldsymbol{x}_j|\boldsymbol{\mu}_l,\boldsymbol{\Sigma}_l\right)},\end{aligned}$$

即给出了样本x_j由第i个高斯混合成分生成的后验概率。

当混合高斯分布已知时，混合高斯聚类算法将把样本集划分为k个簇 $C = C_1, C_2, \ldots, C_k$，每个样本$x_j$的簇标记$\lambda_j$如下式确定：

$$\lambda_j = \underset{i \in \{1,2,\cdots,k\}}{\arg\max}\ \gamma_{ji}.$$

对于给定样本集及模型参数$\{(\alpha_i, \mu_i, \Sigma_i) | 1 \le i \le k\}$的求解，可以采用极大似然估计，即最大化对数似然函数：

$$\begin{aligned} LL(D) &= \ln\left(\prod_{j=1}^{m} p_{\mathcal{M}}(\boldsymbol{x}_j)\right) \\ &= \sum_{j=1}^{m} \ln\left(\sum_{i=1}^{k} \alpha_i \cdot p(\boldsymbol{x}_j | \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)\right) \end{aligned}$$

EM 算法：

1．EM算法是含有隐变量的概率模型极大似然估计或极大后验概率估计的迭代算法。含有隐变量的概率模型的数据表示为θ）。这里，Y是观测变量的数据，Z是隐变量的数据，θ是模型参数。EM算法通过迭代求解观测数据的对数似然函数$L(\theta) = \log P(\mathrm{Y}|\theta)$的极大化，实现极大似然估计。每次迭代包括两步：

E步，求期望，即求$logP(Z|Y,\theta)$关于$P(Z|Y,\theta^{(i)})$的期望：

$$Q(\theta, \theta^{(i)}) = \sum_{Z} \log P(Y, Z|\theta) P(Z|Y, \theta^{(i)})$$

称为Q函数，这里$\theta^{(i)}$是参数的现估计值；

M步，求极大，即极大化Q函数得到参数的新估计值：

$$\theta^{(i+1)} = \arg\max_{\theta} Q(\theta, \theta^{(i)})$$

在构建具体的EM算法时，重要的是定义Q函数。每次迭代中，EM算法通过极大化Q函数来增大对数似然函数$L(\theta)$。

2．EM算法在每次迭代后均提高观测数据的似然函数值，即

$$P(Y|\theta^{(i+1)}) \geqslant P(Y|\theta^{(i)})$$

在一般条件下EM算法是收敛的，但不能保证收敛到全局最优。

E 步：

$$\mu^{i+1} = \frac{\pi(p^i)^{y_i}(1-(p^i))^{1-y_i}}{\pi(p^i)^{y_i}(1-(p^i))^{1-y_i} + (1-\pi)(q^i)^{y_i}(1-(q^i))^{1-y_i}}$$

M 步：

$$\pi^{i+1} = \frac{1}{n}\sum_{j=1}^{n} \mu_j^{i+1}$$

$$p^{i+1} = \frac{\sum_{j=1}^{n} \mu_j^{i+1} y_i}{\sum_{j=1}^{n} \mu_j^{i+1}}$$

$$q^{i+1} = \frac{\sum_{j=1}^{n} (1-\mu_j^{i+1} y_i)}{\sum_{j=1}^{n} (1-\mu_j^{i+1})}$$

本次实验中，直接以 k-means 的结果作为 GMM 的初始中心点，再应用 EM 算法对其优化，实验效果显著。

伪代码如下：

Algorithm 2 Gaussian mixture clustering algorithm

Input: $D=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_m\}$, 高斯混合成分个数 k

Output: 簇划分 $\mathcal{C}=\{C_1,C_2,\cdots,C_k\}$

初始化高斯混合分布的模型参数 $\{(\alpha_i,\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i)\,|1\leq i\leq k\}$
repeat
  for $j=1,2,\cdots,m$ **do**
    根据式 (4) 计算计算 $\gamma_{ji}=p_{\mathcal{M}}\left(z_j=i|\boldsymbol{x}_j\right)(1\leq i\leq k)$
  end for
  for $i=1,2,\cdots,k$ **do**
    计算新均值向量：$\boldsymbol{\mu}_i'=\frac{\sum_{j=1}^{m}\gamma_{ji}\boldsymbol{x}_j}{\sum_{j=1}^{m}\gamma_{ji}}$;
    计算新协方差矩阵：$\boldsymbol{\Sigma}_i'=\frac{\sum_{j=1}^{m}\gamma_{ji}(\boldsymbol{x}_j-\boldsymbol{\mu}_i')(\boldsymbol{x}_j-\boldsymbol{\mu}_i')^{\mathrm{T}}}{\sum_{j=1}^{m}\gamma_{ji}}$;
    计算新混合系数：$\alpha_i'=\frac{\sum_{j=1}^{m}\gamma_{ji}}{m}$;
  end for
  将模型参数 $\{(\alpha_i,\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i)\,|1\leq i\leq k\}$ 更新为 $\{(\alpha_i',\boldsymbol{\mu}_i',\boldsymbol{\Sigma}_i')\,|1\leq i\leq k\}$
until 满足停止条件
$C_i=\varnothing\,(1\leq i\leq k)$
for $j=1,2,\cdots,m$ **do**
  根据式 (5) 确定 $\boldsymbol{x}_j$ 的簇标记 λ_j;
  将 $\boldsymbol{x}_j$ 划入相应的簇：$C_{\lambda_j}=C_{\lambda_j}\cup\{\boldsymbol{x}_j\}$
end for

具体实现如下：

```
def __init_params(self):
      # 以k means结果作为初始中心点
      # 初始化协方差sigma
      sigma = collections.defaultdict(list)
      for i in range(self.k):
            sigma[i] = np.eye(self.data_columns, dtype=float) * 0.1
      return sigma

def __likelihood(self):
      likelihood = np.zeros((self.data_rows, self.k))
      for i in range(self.k):
            likelihood[:, i] = multivariate_normal.pdf(self.data, self.mu[i], self.sigma[i])
      return likelihood

def __expectation(self):
      # 求期望 E
      likelihood = self.__likelihood() * self.alpha # (m,k)
      sum_likelihood = np.expand_dims(np.sum(likelihood, axis=1), axis=1) # (m,1)
        print(np.log(np.prod(sum_likelihood)))        # 输出似然值
      self.gamma = likelihood / sum_likelihood
      # print(self.gamma)
      self.data_attribution = self.gamma.argmax(axis=1)
      for i in range(self.data_rows):
            self.c[self.data_attribution[i]].append(self.data[i].tolist())

def __maximization(self):
      # 最大化 M
      for i in range(self.k):
            gamma_ji = np.expand_dims(self.gamma[:, i], axis=1)       # 提取每一列 作为列向量 (m, 1)
            mu_i = (gamma_ji * self.data).sum(axis=0) / gamma_ji.sum()
            cov = (self.data - mu_i).T.dot((self.data - mu_i) * gamma_ji) / gamma_ji.sum()
            self.mu[i], self.sigma[i] = mu_i, cov       # 更新参数
      self.alpha = self.gamma.sum(axis=0) / self.data_rows
```

收敛条件为各参数变化小于给定精度：

```
def gmm(self):
    pre_alpha = self.alpha
    pre_mu = self.mu
    pre_sigma = self.sigma
    for i in range(self.iteration):
        self.__expectation()
        self.__maximization()
        diff = np.linalg.norm(pre_alpha - self.alpha)\
        + np.linalg.norm(pre_mu - self.mu)\
        + np.sum([np.linalg.norm(pre_sigma[i] - self.sigma[i]) for i in range(self.k)])
        if diff > self.eps:
            pre_alpha = self.alpha
            pre_sigma = self.sigma
            pre_mu = self.mu
        else: # 迭代终止条件 参数sigma mu和alpha几乎不变化
            break
    self.__expectation()
    return self.mu, self.c
```

3.3 生成数据

以给定的各类别中心，分别生成对应个数的样本点：

```
def generate_data(sample_means, sample_num, k_num):
    """ 生成2维数据
    :argument sample_means k类数据的均值 以list的形式给出 如[[1, 2],[-1, -2], [0, 0]]
    :argument sample_num k类数据的数量 以list的形式给出 如[10, 20, 30]
    :argument k_num k类
    """
    assert k_num > 0
    assert len(sample_means) == k_num
    assert len(sample_num) == k_num

    cov = [[0.1, 0], [0, 0.1]]
    data = []
    for i in range(k_num):
        for k in range(sample_num[i]):
            data.append(np.random.multivariate_normal([sample_means[i][0], sample_means[i][1]], cov).tolist())
    return np.array(data)
```

3.4 结果可视化

思想较为简单，不作赘述，看代码及注释即可：

```
def show(k, mu, c, title):
    plt.title(title)
    for i in range(k):
        plt.scatter(np.array(c[i])[:, 0], np.array(c[i])[:, 1], label=str(i + 1))
    plt.scatter(mu[:, 0], mu[:, 1], c="b", label="center")
    plt.legend()
    plt.show()
```

3.5 UCI 数据集处理

文件处理不做赘述，其中测试聚类准确性的函数可复用于 k-means 及 GMM 算法的结果：

```
class IrisProcessing(object):
      def __init__(self):
            self.data_set = pd.read_csv("./data/iris.csv")
            self.x = self.data_set.drop('class', axis=1)
            self.y = self.data_set['class']
            self.classes = list(it.permutations(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], 3))

      def read(self):
            return np.array(self.x, dtype=float)

      def cal_accuracy(self, y_label):
            """ 用于测试聚类的正确率 """
            num = len(self.y)
            counts = []
            for i in range(len(self.classes)):
                  count = 0
                  for j in range(num):
                        if self.y[j] == self.classes[i][y_label[j]]:
                              count += 1
                  counts.append(count)
            return np.max(counts) * 1.0 / num
```

四、 实验结果与分析

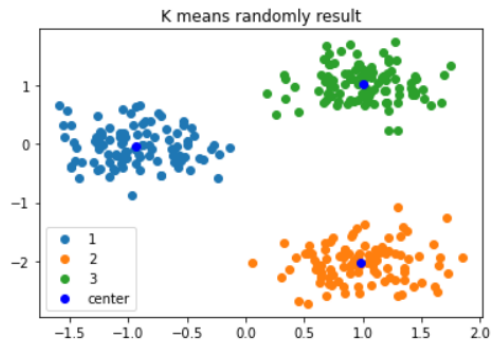
4.1 人工生成距离较远的数据集：

```
k = 3
means = [[1, 1], [-1, 0], [1, -2]]
number = [100, 100, 100]
data = generate_data(means, number, k)
```

4.1.1 k-means 结果

```
km = k_means.KMeans(data, k)
random_mu, random_c = km.k_means_random_center()
show(k, random_mu, random_c, "K means randomly result")
```

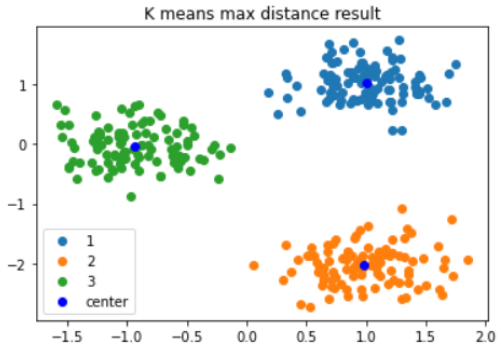
```
K-means 1
K-means 2
K-means 3
K-means 4
```



随机初始聚类中心

```
max_mu, max_c = km.k_means_not_random_center()
show(k, max_mu, max_c, "K means max distance result")
```

```
K-means 1
K-means 2
```

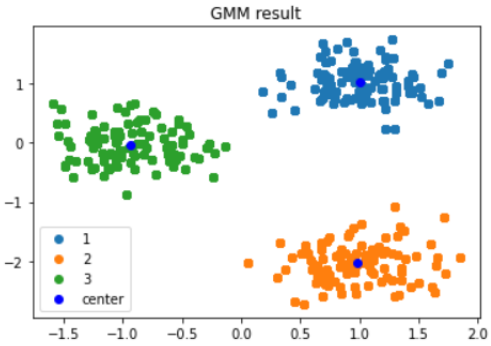


距离最大化初始聚类中心

可以看到，距离最大化初始聚类中心的迭代次数为 2，比随机初始聚类中心迭代次数少，因为距离最大化初始聚类中心是最可能保证初始中心均不在同一类。

4.1.2 GMM 结果

```
eps = 1e-12
iteration = 10000
gmm = GMM.GaussianMixtureModel(data, max_mu, k, eps, iteration)
gmm_mu, gmm_c = gmm.gmm()
show(k, gmm_mu, gmm_c, "GMM result")
```



由于数据较为分散，各算法差别不大。

4.2 人工生成距离较近的数据集

```
k = 5
means = [[2,2],[2,4],[5,5],[8,7],[8,8]]
number = [50, 50, 50, 50, 50]
data = generate_data(means, number, k)
km = k_means.KMeans(data, k)
random_mu, random_c = km.k_means_random_center()
show(k, random_mu, random_c, "K means randomly result")
max_mu, max_c = km.k_means_not_random_center()
show(k, max_mu, max_c, "K means max distance result")
eps = 1e-12
iteration = 10000
gmm = GMM.GaussianMixtureModel(data,max_mu, k,eps,iteration)
gmm_mu, gmm_c = gmm.gmm()
show(k, gmm_mu, gmm_c, "GMM result")
```

4.2.1 k-means

```
K-means 1
K-means 2
K-means 3
K-means 4
K-means 5
K-means 6
K-means 7
```

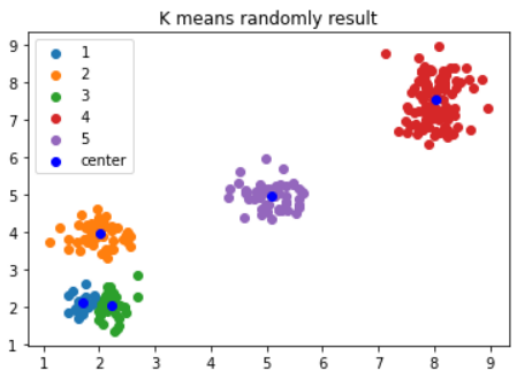


图 1 随机初始聚类中心

```
K-means 1
K-means 2
K-means 3
K-means 4
K-means 5
K-means 6
```

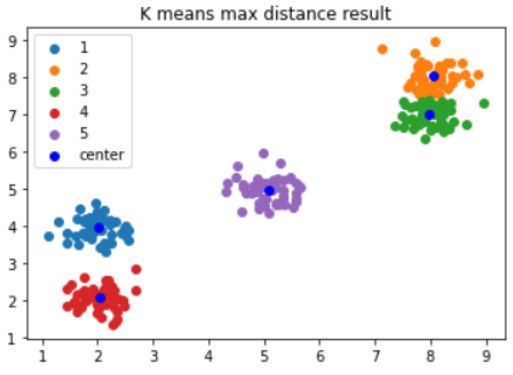
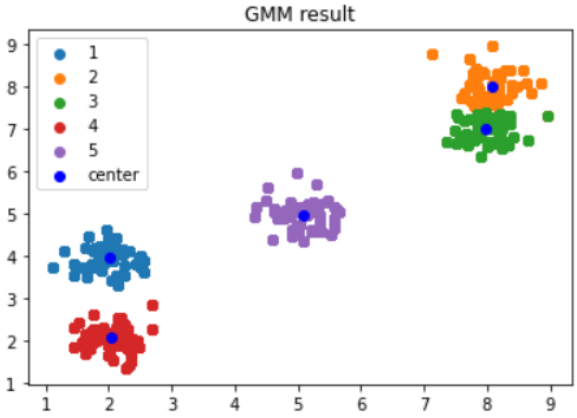


图 2 距离最大化初始聚类中心

可以看到，距离最大化初始聚类中心的迭代次数为 6，比随机初始聚类中心迭代次数少； 首先采用随机选取初始簇中心的方法，实验结果如图 1 所示，很明显在分类上出现了错误，分析其原因，为没有选好初始簇中心，导致了聚类时无法区分与簇中心较远而距离大致相近的点；第二次选择了选择相距尽可能远的点作为初始的簇中心，聚类结果如图 2 所示，此次初始簇中心相距较远，较好的划分了样本集。

4.2.2 GMM



在 k-means 结果上进行优化，更为准确。

4.3 UCI 数据集

选用 UCI 鸢尾花分类数据集，根据鸢尾花的 4 个属性对鸢尾花的分类进行预测. 数据集中一共有三种类别，每个类别各 50 个样本，一共 150 个样本，每条数据包括四个

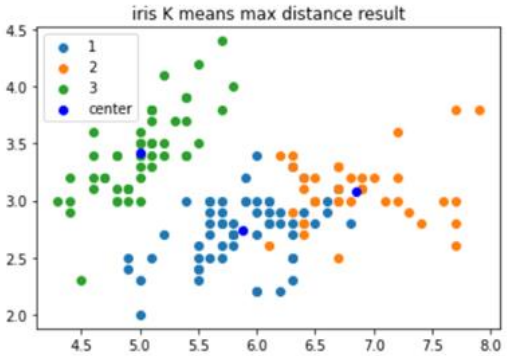
特征，分别是：

1. 萼片长度（单位：厘米）
2. 萼片宽度（单位：厘米）
3. 花瓣长度（单位：厘米）
4. 花瓣宽度（单位：厘米）

4.3.1 k-means

选择距离最大化初始聚类中心

```
K-means 1
K-means 2
K-means 3
K-means 4
K-means 5
K-means 6
K-means 7
K-means 8
K-means 9
K-means 10
K-means 11
K-means 12
K-means 13
[[5.88360656 2.74098361 4.38852459 1.43442623]
 [6.85384615 3.07692308 5.71538462 2.05384615]
 [5.006      3.418      1.464      0.244     ]]
```

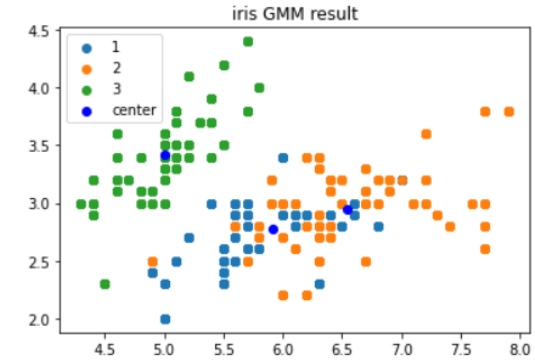



```
0.8866666666666667
```

迭代次数为 13 次，准确率为 88.67%

4.3.2 GMM

```
[[5.91496959 2.77784365 4.20155323 1.29696685]
 [6.54454865 2.94866115 5.47955343 1.98460495]
 [5.006      3.418      1.464      0.244     ]]
```




```
0.9666666666666667
```

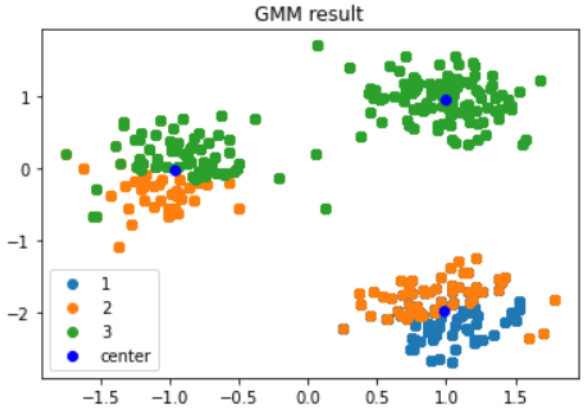
准确率高达 96.67%，相比 k-means 提升了 8%

可以看到，使用 GMM 算法，对于每个类别聚类效果都比较好。并且通过准确率也可以看出，我们得到的模型实际上是接近于真实的数据分布的。需要注意的是，这里 GMM 参数的初始化使用了 k-means 算法的结果。

若 GMM 算法随机初始化聚类中心：
将算法代码修改如下：

```
class GaussianMixtureModel(object):
    """ 高斯混合聚类EM算法 """

    def __init__(self, data, mu_kmeans, k, eps=1e-12, iteration=10000):
        self.data = data
        self.k = k
        self.eps = eps
        self.iteration = iteration
        self.data_rows = self.data.shape[0]
        self.data_columns = self.data.shape[1]
        self.alpha = np.ones(self.k) * (1.0 / self.k)
#           self.mu = mu_kmeans
        # 随机选择k个点作为初始点 极易陷入局部最小值
        self.mu = np.array(self.data[random.sample(range(self.data_rows), self.k)])
        self.sigma = self.__init_params()
        self.data_attribution = [-1] * self.data_rows
        self.c = collections.defaultdict(list)
        self.gamma = None
```



当随机初始化时，模型的准确率会变得不稳定，说明此时模型落入了局部最优解。因此 GMM 算法对于参数的初始值极度敏感，在结论部分我们会进一步讨论这一点。

4.4 空簇问题的解决

在实验过程中，经常会出现产生空簇的情况，这实际上是模型退化或者落入局部最优解的现象。经过查阅资料找到以下解决方法：

✓ 调整 K 值

在空簇的相邻簇中挑出一部分给该簇并继续迭代，直至下一次收敛

五、 结论

- ✧ K-Means 聚类假设数据分布在以簇中心为中心的一定范围内，混合高斯模型则假设数据符合混合高斯分布；
- ✧ K-Means 假设使用的欧式距离来衡量样本与各个簇中心的相似度(假设数据的各个维度对于相似度计算的作用是相同的)；
- ✧ GMM 使用 EM 算法进行迭代优化，因为其涉及到隐变量的问题，没有之前的完全数据，而是在不完全数据上进行；
- ✧ K-Means 聚类假设数据分布在以簇中心为中心的一定范围内，混合高斯模型则假设数据符合混合高斯分布；
- ✧ 由于 K-Means 聚类算法采用贪心的思想，未必能得到全局最优解，簇中心初始化对于最终的结果有很大的影响，如果选择不好初始的簇中心值容易使之陷入局部最优解

六、 参考文献

[1] 李航, 《统计学习方法》(第三版)

[2] 周志华, 《机器学习》

[3] Pattern Recognition and Machine Learning.

[4] Iris Data Set. (1988.7) [Data set]

七、 附录：源代码（带注释）

k_means.py

```
import numpy as np

import random

import collections

from scipy.stats import multivariate_normal


def cal_dis(x1,x2):
```

```
    return np.linalg.norm(x1-x2)


class GaussianMixtureModel(object):
    """ 高斯混合聚类 EM 算法 """


    def __init__(self, data, mu_kmeans, k, eps=1e-12, iteration=1000
0):
        self.data = data
        self.k = k
        self.eps = eps
        self.iteration = iteration
        self.data_rows = self.data.shape[0]
        self.data_columns = self.data.shape[1]
        self.alpha = np.ones(self.k) * (1.0 / self.k)
        self.mu = mu_kmeans
        # 随机选择 k 个点作为初始点 极易陷入局部最小值
        # self.mu = np.array(self.data[random.sample(range(self.data_ro
ws), self.k)])
        self.sigma = self.__init_params()
        self.data_attribution = [-1] * self.data_rows
        self.c = collections.defaultdict(list)
        self.gamma = None


    def __init_params(self):
        # 以 k means 结果作为初始中心点
        # 初始化协方差 sigma
        sigma = collections.defaultdict(list)
        for i in range(self.k):
            sigma[i] = np.eye(self.data_columns, dtype=float) * 0.1
        return sigma
```

```
    def __likelihood(self):
        likelihood = np.zeros((self.data_rows, self.k))
        for i in range(self.k):
            likelihood[:, i] = multivariate_normal.pdf(self.data, self.
mu[i], self.sigma[i])
        return likelihood


    def __expectation(self):
        # 求期望 E
        likelihood = self.__likelihood() * self.alpha # (m,k)
        sum_likelihood = np.expand_dims(np.sum(likelihood, axis=1), axi
s=1) # (m,1)
#         print(np.log(np.prod(sum_likelihood)))     # 输出似然值
        self.gamma = likelihood / sum_likelihood
        # print(self.gamma)
        self.data_attribution = self.gamma.argmax(axis=1)
        for i in range(self.data_rows):
            self.c[self.data_attribution[i]].append(self.data[i].tolist
())


    def __maximization(self):
        # 最大化 M
        for i in range(self.k):
            gamma_ji = np.expand_dims(self.gamma[:, i], axis=1)    # 提取
每一列 作为列向量 (m, 1)
            mu_i = (gamma_ji * self.data).sum(axis=0) / gamma_ji.sum()
            cov = (self.data - mu_i).T.dot((self.data - mu_i) * gamma_j
i) / gamma_ji.sum()
            self.mu[i], self.sigma[i] = mu_i, cov    # 更新参数
        self.alpha = self.gamma.sum(axis=0) / self.data_rows
```

```
    def gmm(self):
        pre_alpha = self.alpha
        pre_mu = self.mu
        pre_sigma = self.sigma
        for i in range(self.iteration):
            self.__expectation()
            self.__maximization()
            diff = np.linalg.norm(pre_alpha - self.alpha)\
            + np.linalg.norm(pre_mu - self.mu)\
            + np.sum([np.linalg.norm(pre_sigma[i] - self.sigma[i]) for i
 in range(self.k)])
            if diff > self.eps:
                pre_alpha = self.alpha
                pre_sigma = self.sigma
                pre_mu = self.mu
            else: # 迭代终止条件 参数 sigma mu 和 alpha 几乎不变化
                break
        self.__expectation()
        return self.mu, self.c
```

GMM.py

```
import numpy as np
import random
import collections
from scipy.stats import multivariate_normal


def cal_dis(x1,x2):
    return np.linalg.norm(x1-x2)
```

```
class GaussianMixtureModel(object):
    """ 高斯混合聚类 EM 算法 """

    def __init__(self, data, mu_kmeans, k, eps=1e-12, iteration=1000
0):
        self.data = data
        self.k = k
        self.eps = eps
        self.iteration = iteration
        self.data_rows = self.data.shape[0]
        self.data_columns = self.data.shape[1]
        self.alpha = np.ones(self.k) * (1.0 / self.k)
        self.mu = mu_kmeans
        # 随机选择 k 个点作为初始点 极易陷入局部最小值
        # self.mu = np.array(self.data[random.sample(range(self.data_ro
ws), self.k)])
        self.sigma = self.__init_params()
        self.data_attribution = [-1] * self.data_rows
        self.c = collections.defaultdict(list)
        self.gamma = None

    def __init_params(self):
        # 以 k means 结果作为初始中心点
        # 初始化协方差 sigma
        sigma = collections.defaultdict(list)
        for i in range(self.k):
            sigma[i] = np.eye(self.data_columns, dtype=float) * 0.1
        return sigma

    def __likelihood(self):
        likelihood = np.zeros((self.data_rows, self.k))
```

```
        for i in range(self.k):
            likelihood[:, i] = multivariate_normal.pdf(self.data, self.mu[i], self.sigma[i])
        return likelihood

    def __expectation(self):
        # 求期望 E
        likelihood = self.__likelihood() * self.alpha # (m,k)
        sum_likelihood = np.expand_dims(np.sum(likelihood, axis=1), axis=1) # (m,1)
#         print(np.log(np.prod(sum_likelihood)))     # 输出似然值
        self.gamma = likelihood / sum_likelihood
        # print(self.gamma)
        self.data_attribution = self.gamma.argmax(axis=1)
        for i in range(self.data_rows):
            self.c[self.data_attribution[i]].append(self.data[i].tolist())

    def __maximization(self):
        # 最大化 M
        for i in range(self.k):
            gamma_ji = np.expand_dims(self.gamma[:, i], axis=1)    # 提取每一列 作为列向量 (m, 1)
            mu_i = (gamma_ji * self.data).sum(axis=0) / gamma_ji.sum()
            cov = (self.data - mu_i).T.dot((self.data - mu_i) * gamma_ji) / gamma_ji.sum()
            self.mu[i], self.sigma[i] = mu_i, cov    # 更新参数
        self.alpha = self.gamma.sum(axis=0) / self.data_rows

    def gmm(self):
        pre_alpha = self.alpha
```

```
        pre_mu = self.mu
        pre_sigma = self.sigma
        for i in range(self.iteration):
            self.__expectation()
            self.__maximization()
            diff = np.linalg.norm(pre_alpha - self.alpha)\
            + np.linalg.norm(pre_mu - self.mu)\
            + np.sum([np.linalg.norm(pre_sigma[i] - self.sigma[i]) for i
 in range(self.k)])
            if diff > self.eps:
                pre_alpha = self.alpha
                pre_sigma = self.sigma
                pre_mu = self.mu
            else: # 迭代终止条件 参数 sigma mu 和 alpha 几乎不变化
                break
        self.__expectation()
        return self.mu, self.c
```

iris_read.py

```
import numpy as np
import pandas as pd
import itertools as it


class IrisProcessing(object):
    def __init__(self):
        self.data_set = pd.read_csv("./data/iris.csv")
        self.x = self.data_set.drop('class', axis=1)
        self.y = self.data_set['class']
        self.classes = list(it.permutations(['Iris-setosa', 'Iris-versi
color', 'Iris-virginica'], 3))
```

```
    def read(self):
        return np.array(self.x, dtype=float)


    def cal_accuracy(self, y_label):
        """ 用于测试聚类的正确率 """
        num = len(self.y)
        counts = []
        for i in range(len(self.classes)):
            count = 0
            for j in range(num):
                if self.y[j] == self.classes[i][y_label[j]]:
                    count += 1
            counts.append(count)
        return np.max(counts) * 1.0 / num
```

k_means.ipynb

```
import numpy as np
from matplotlib import pyplot as plt
from lab3 import GMM
from lab3 import k_means
from lab3 import iris_read
def generate_data(sample_means, sample_num, k_num):
    """ 生成 2 维数据
    :argument sample_means k 类数据的均值 以 list 的形式给出 如[[1, 2],[-1,
-2], [0, 0]]
    :argument sample_num k 类数据的数量 以 list 的形式给出 如[10, 20, 30]
    :argument k_num k 类
    """
    assert k_num > 0
    assert len(sample_means) == k_num
```

```
    assert len(sample_num) == k_num


    cov = [[0.1, 0], [0, 0.1]]
    data = []
    for i in range(k_num):
        for k in range(sample_num[i]):
            data.append(np.random.multivariate_normal([sample_means[i]
[0], sample_means[i][1]], cov).tolist())
    return np.array(data)
def show(k, mu, c, title):
    plt.title(title)
    for i in range(k):
        plt.scatter(np.array(c[i])[:, 0], np.array(c[i])[:, 1], label=s
tr(i + 1))
    plt.scatter(mu[:, 0], mu[:, 1], c="b", label="center")
    plt.legend()
    plt.show()
k = 3
means = [[1, 1], [-1, 0], [1, -2]]
number = [100, 100, 100]
data = generate_data(means, number, k)
km = k_means.KMeans(data, k)
mu, c = km.k_means_random_center()
show(k, mu, c, "K means randomly result")
max_mu, max_c = km.k_means_not_random_center()
show(k, max_mu, max_c, "K means max distance result")
eps = 1e-12
iteration = 10000
gmm = GMM.GaussianMixtureModel(data,max_mu, k,eps,iteration)
gmm_mu, gmm_c = gmm.gmm()
show(k, gmm_mu, gmm_c, "GMM result")
```

```
k = 5

means = [[2,2],[2,4],[5,5],[8,7],[8,8]]

number = [50, 50, 50, 50, 50]

data = generate_data(means, number, k)

km = k_means.KMeans(data, k)

random_mu, random_c = km.k_means_random_center()

show(k, random_mu, random_c, "K means randomly result")

max_mu, max_c = km.k_means_not_random_center()

show(k, max_mu, max_c, "K means max distance result")

eps = 1e-12

iteration = 10000

gmm = GMM.GaussianMixtureModel(data,max_mu, k,eps,iteration)

gmm_mu, gmm_c = gmm.gmm()

show(k, gmm_mu, gmm_c, "GMM result")

iris = iris_read.IrisProcessing()

iris_data= iris.read()

eps = 1e-12

iteration = 10000

km_iris = k_means.KMeans(iris_data, 3)

km_mu_iris, km_c_iris = km_iris.k_means_not_random_center()

print(km_mu_iris)

show(3, km_mu_iris, km_c_iris, "iris K means max distance result")

print(iris.cal_accuracy(km_iris.data_attribution))

gmm_iris = GMM.GaussianMixtureModel(iris_data,km_mu_iris, 3 ,eps, iter
ation)

gmm_mu_iris, gmm_c_iris = gmm_iris.gmm()

print(gmm_mu_iris)

show(3, gmm_mu_iris, gmm_c_iris, "iris GMM result")

print(iris.cal_accuracy(gmm_iris.data_attribution))
```