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	余涛
学号	1180300829
班号	1803202
电子邮件	1063695334@qq.com
手机号码	15586430583

目录

1 实验内容

1.1 实验目的

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

1.2 实验要求

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
可以获得正确结果(与你的设定结果比较)。
应用：
可以在 UCI 上找一个简单问题数据，用你实现的 GMM 进行聚类。

1.3 实验环境

Windows 10 专业版；python 3.8.6；PyCharm Community Edition 2020.2.2 x64

2 实验设计思想

2.1 算法原理

2.1.1 选取彼此距离尽可能大的向量作为初始的均值向量中心

对于给定的样本集，需要选取样本中彼此距离尽可能远的 k 个向量作为初始的均值向量，这样能避免出现糟糕的分类结果。具体思路为定义一个储存初值向量的集合，每次选取欧式距离到这个集合最大的向量加入到均值向量集合中，最后返回均值向量中心集合。

2.1.2 EM 算法原理

两个算法 K-Means 和 GMM 的实现本质上都是 EM 算法的应用。所以先介绍 EM 算法的思想：

EM 算法分为两步，

E 步：expectation step，求期望步

M 步：maximization step，最大化似然步

其中：

E 步是初始化参数和调整分布，将所有样本点进行聚类分布。

M 步是根据调整的分布，求使得目标函数最大化的参数，从而更新了参数。

接着参数的更新又可以调整分布，不断循环，直到参数的变化收敛，这一过程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。

2.1.3 K-Means 算法原理

K-Means 算法是 EM 算法的一个简单体现，K-Means 的伪码如下：

```
k ← 输入期望的聚类数
centers ← 从样本中初始化 k 个聚类中心（可以随机选取）
while true:
    E 步：遍历所有样本，根据样本到 k 个聚类中心的距离度量，判断该样本的标签（类别）
labels[i]
    M 步：根据标签划分结果 labels，重新计算 k 个聚类中心（centers）
    if k 个聚类中心的变化全部小于误差值 eps:
     break
return centers, labels
```

给定样本集 D={x1,x2,⋯,xm}和划分聚类的数量 k，然后给定一个簇划分 C={C1,C2,⋯,Ck}，为了使得该簇划分的平方误差 E 最小化，平方误差 E 的表达式为：

$$E = \sum_{i=1}^{k} \sum_{\mathbf{x} \in \mathbf{C_i}} \|\mathbf{x} - \mu_{\mathbf{i}}\|_2^2 \qquad (1)$$

在该式子中$\mu_{\mathbf{i}} = \frac{1}{|\mathbf{C_i}|}\sum_{\mathbf{x} \in \mathbf{C_i}}$ x 是簇 Ci 的均值向量。并且 E 能刻画出簇内样本的内聚紧密程度，如果 E 越小，则说明该簇内样本的相似度越高。

思路如下：

由于实现了选取彼此距离尽可能大的向量作为均值向量，所以我们能够避免初始化的均值向量距离过近而导致糟糕的分类结果。选取初值后进行不断的迭代，迭代主要分为两步，然后开始不断迭代，迭代主要分为两步,这两步分别对应着 EM 算法的 E 步和 M 步。其中 E 步是进行标签的划分，即为样本集进行分类，具体做法为遍历 D 中的所有点，对任意一个点计

算它到 3 个聚类中心的欧式距离，取距离最小的聚类均值中心所代表的那一类，作为该点的标签。经过 E 步过后，所有样本都更新了标签，然后进入 M 步，根据 E 步更新的标签，得到了每一类的样本，然后分别对每个聚类，计算新的均值向量中心，然后用得到的新的均值向量中心作为下一轮迭代的初始均值向量中心，直到每个均值向量中心与前变换前的均值向量中心相等。

具体的迭代优化策略如下：

1. 首先初始化一组均值向量（选取彼此距离尽可能大的向量作为均值向量）。
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，假设正确；否则，将真实的均值向量作为新的假设均值向量，然后回到 1.继续迭代求解。

2.1.4 GMM 算法原理

GMM 相对 K-Means 来说是比较复杂的 EM 算法的应用实现。具体的数学原理如下：

首先需要给出维样本空间中的随机变量服从高斯分布的密度函数：

$$p(\mathbf{x} \mid \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right) \quad (2)$$

参数中，$\mu = \{\mu_1, \mu_2, \dots, \mu_n\}$为 n 维的均值向量，Σ为 n×n 的协方差矩阵。

需要给出混合高斯分布的定义：

$$p_{\mathcal{M}} = \sum_{i=1}^{k} \alpha_i p(\mathbf{x} \mid \mu_\mathbf{i}, \Sigma_i) \quad (3)$$

对于这个分布来说，由 k 个高斯分布成分混合构成。其中μ_1，Σi 是第 i 个高斯分布的均值和协方差矩阵，$\alpha_i > 0$为对应的混合系数，且满足$\sum_{i=1}^{k} \alpha_i = 1$。

假设样本集 D 由高斯混合分布给出，首先根据α_1，α_2,⋯,α_k所定义的先验分布选择高斯分布混合成分，即$p(z_j = i) = \alpha_i$，其中$z_j \in \{1,2, \dots, k\}$，然后根据被选择的高斯混合成分的概率密度函数进行采样，从而生成相应的样本。

根据贝叶斯定理，z_j的后验分布对应于：

$$p_{\mathcal{M}}(z_j = i \mid \mathbf{x_j}) = \frac{p(z_j=i)\cdot p_{\mathcal{M}}(\mathbf{x_j}|z_j=i)}{p_{\mathcal{M}}(\mathbf{x_j})} = \frac{\alpha_i \cdot p(\mathbf{x_j}|\mu_\mathbf{i},\Sigma_i)}{\sum_{l=1}^{k}\alpha_l p(\mathbf{x_j}|\mu_\mathbf{l},\Sigma_l)} \quad (4)$$

其中$p_{\mathcal{M}}(z_j = i \mid \mathbf{x_j})$给出了样本 $\mathbf{x_j}$由第 i 个高斯分布生成的后验概率。

当式(3)已知时，混合高斯模型将样本集 D 划分成了 k 个簇$C = \{\mathbf{C_1}, \mathbf{C_2}, \dots, \mathbf{C_k}\}$，然后对于每个样本$\mathbf{x_j}$，将其簇标记为$\lambda_\mathbf{j}$：

$$\lambda_j = \arg max i p_{\mathcal{M}}(z_j = i \mid \mathbf{x_j}) \quad (5)$$

因此对于 GMM 算法来说，关键在于三个参数$\{\boldsymbol{\alpha_i}, \boldsymbol{\mu_i}, \boldsymbol{\Sigma_i} \mid \boldsymbol{i} \in \{\mathbf{1}, \mathbf{2}, \dots, \boldsymbol{k}\}\}$的求解。

当给定样本集 D 后，可以采用极大似然估计来估计参数，即最大化对数似然：

$$LL(D) = \ln\left(\prod_{j=1}^{m} p_{\mathcal{M}}(\mathbf{x_j})\right) = \sum_{j=1}^{m} \ln\left(\sum_{i=1}^{k} \alpha_i \cdot p(\mathbf{x_j} \mid \mu_i, \Sigma_i)\right) \quad (6)$$

为了让(6)最大化，对μ_i求导，然后令导数为 0 可以得到：

$$\sum_{j=1}^{m} \frac{\alpha_i \cdot p(\mathbf{x_j}|\mu_i,\Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu l,\Sigma_l)} \Sigma_i^{-1}(\mathbf{x_j} - \mu_i) = 0 \quad (7)$$

然后对于两边同时乘Σ_i，进行化简可以得到：

$$\mu_i = \frac{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j=i|\mathbf{x_j}) \cdot \mathbf{x_j}}{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j=i|\mathbf{x_j})} \quad (8)$$

这样就说明各个混合成分的均值可以通过样本加权平均估计，权重样本式是每个样本属于该成分的后验概率。

然后同理对(6)式来说，对Σ_i求导，然后令导数为 0 可以得到：

$$\Sigma_i = \frac{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j=i|\mathbf{x_j}) \cdot (\mathbf{x_j}-\mu_i)(\mathbf{x_j}-\mu_i)^T}{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j=i|\mathbf{x_j})} \quad (9)$$

对于混合系数α_i，其还需要满足$\alpha_i \geq 0, \sum_i^k \alpha_i = 1$，所以在式(6)的基础上增加拉格朗日项：

$$LL(D) + \lambda\left(\sum_{i=1}^{k} \alpha_i - 1\right) \quad (10)$$

其中λ是拉格朗日乘子，由式(10)，对于α_i求导，然后令导数为 0 可以得到：

$$\sum_{j=1}^{m} \frac{p(\mathbf{x_j}|\mu_i,\Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu_l,\Sigma_l)} + \lambda = 0 \quad (11)$$

对于式(11)，两边同乘α_i，然后将$\mathrm{i} \in \{1,2,\ldots,k\}$代入进行相加可以得到：

$$\sum_{i=1}^{k} \left(\alpha_i \cdot \sum_{j=1}^{m} \frac{p(\mathbf{x_j}|\mu_i,\Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu_i,\Sigma_i)}\right) + \lambda \sum_{i=1}^{k} \alpha_i = 0 \quad (12)$$

进行整理，由于$\sum_{i=1}^{k} \alpha_i = 1$，有：

$$\sum_{j=1}^{m} \left(\frac{\sum_{i=1}^{k} \alpha_i \cdot p(\mathbf{x_j}|\mu_i,\Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu_i,\Sigma_i)}\right) + \lambda = m + \lambda = 0 \quad (13)$$

从而有$\lambda = -m$，然结合式(11)有：

$$\alpha_i = \frac{1}{m} \sum_{j=1}^{m} \frac{p(\mathbf{x_j}|\mu_i,\Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu_l,\Sigma_l)} \quad (14)$$

可见，每个高斯混合系数由样本属于该成分的平均后验概率决定。

式子(8)(9)(14)分别给出了三个参数$\{\boldsymbol{\alpha_i}, \boldsymbol{\mu_i}, \boldsymbol{\Sigma_i} \mid \boldsymbol{i} \in \{\mathbf{1}, \mathbf{2}, \ldots, \boldsymbol{k}\}\}$的求解方法。

2.2 算法实现

2.2.1 选取彼此距离尽可能大的向量作为初始的均值向量中心

作用：

```
'''
选择彼此距离尽可能远的 K 个点作为初始簇中心点集合{u1, u2,...,uk},储存了所有的均值
向量点
'''
```

具体实现：

首先在 k 个点中随机选出第一个均值初始点，然后，将其加入初始簇中心点集合（k 集合）中，然后遍历样本中的所有数据，选取距离到 k 集合最大的点加入 k 集合中，这样便能找到距离尽可能远的 k 个向量作为初始均值向量点。

具体以注释形式给出。

```
def initial_cluster_center_point_by_maxlength(self):
    mu_0 = np.random.randint(0, self.k) + 1  # 首先在 k 个点中随机选第一个初始点
    mu_collection = [self.data[mu_0]]  # 定义一个初始簇中心点集合（k 集合），将选取的第一个点放入其中
    for m in range(self.k - 1):  # 除了第一个点还需要选取 k-1 个点，每次选择一个距离最大的点加入到 k 集合中
        all_length = []
        for i in range(self.data_rows):  # 对于样本集中的所有数据，求得其到 k 集合的距离

all_length.append(np.sum([self.distance_by_euclidean(self.data[i], mu_collection[j])for j in range(len(mu_collection))]))
        mu_collection.append(self.data[np.argmax(all_length)])  # 取距离最大的点下标加入 k 集合
    print('初始均值向量集合为：\n', np.array(mu_collection))
    return np.array(mu_collection)
```

2.2.2 K-Means 算法实现

作用：

```
'''
k_means 算法的实现，实现对样本的聚类
'''
```

具体实现：

首先从样本集 D 中选择 k 个样本作为初始化的均值向量$\{\mu_1, \mu_2, \ldots, \mu_n\}$，然后不断进行迭代直到收敛。

迭代的过程为：

（1）初始化所有簇$C_i = \emptyset, i = 1,2,\ldots,k$

（2）对$\mathbf{x_j}, j = 1,2,\ldots,$ m 标记为λ_j，使得$\lambda_j = \arg mini\|x_j - \mu_i\|$，即使得每个$\mathbf{x_j}$都属于距离其

最近的均值向量所在的簇

（3）将样本$\mathbf{x_j}$划分到距离最近均值向量的簇$\mathbf{C}_{\lambda_j} = \mathbf{C}_{\lambda_j} \cup \{\mathbf{x_j}\}$

（4）再重新计算每个簇的均值向量$\hat{\mu_i} = \frac{1}{|\mathbf{C_i}|}\sum_{\mathbf{x}\in\mathbf{C_i}} \mathbf{x}$

（5）对于所有的$i \in 1,2,\ldots,k$，若都有$\hat{\mu}_\mathbf{i} = \mu_\mathbf{i}$，则停止迭代。若有不相等的则重新赋值$\mu_\mathbf{i} = \hat{\mu}_\mathbf{i}$继续迭代。

具体以注释形式给出。

```
def the_method_of_kmeans(self):
    number_of_times = 0  # 循环次数
    flag = 0  # 设置循环结束标签
    while True:
        c = collections.defaultdict(list)  # 初始化 Ci=空，
i=1,2,3,...,k，C 为最终的初始均值向量集
        for i in range(self.data_rows):  # 对数据集中所有的点
            dij = [self.distance_by_euclidean(self.data[i],
self.mu_datas[j]) for j in
                   range(self.k)]  # 对初始簇中心点集合中的所有均值向量点 u1，
u2,...,uk,求得两者的||xi-uj||
            lambda_j = np.argmin(dij)  # 求得数据集中最小的距离的簇的下标
            c[lambda_j].append(self.data[i].tolist())  # 将第 i 个点划分到
C-lambda-j 对应的簇中
            self.sample_assignments[i] = lambda_j
        new_mu = np.array([np.mean(c[i], axis=0).tolist() for i in
range(self.k)])  # 求解每个簇的均值作为新的均值向量
        flag = 0  # 初始化标签为 0
        for m in range(self.k):  # 对于所有的 i 属于 1,2,3,...,k，需要所有新
的 ui 都等于旧的 ui,由于是浮点数，所以只需要两次的 ui 误差小于给定误差即可
            if self.distance_by_euclidean(self.mu_datas[m],
                                          new_mu[m]) > self.error:  # 若大于给
定误差，则将新的 ui 赋值给旧的 ui，并置 flag 为 1
                self.mu_datas[m] = new_mu[m]
                flag = 1
        if flag == 0:  # 当没有新的 ui 产生后循环退出
            break
        print('kmeans 得到的均值向量的循环次数为:', number_of_times)
        number_of_times += 1
        print('得到的均值向量集合为: \n', self.mu_datas)
    return self.mu_datas, c
```

2.2.3 GMM 算法实现

作用：

```
'''
GMM 算法，实现对样本的聚类
'''
```

具体实现：对于 GMM 算法，采用 EM 算法进行迭代优化求解：

E 步：先根据当前的参数来计算样本集 D 中所有样本属于各个高斯分布的后验概率。

M 步：根据公式(8)(9)(14)来更新参数，若参数值不再发生变化或者迭代次数最大时则退出迭代。

具体做法为：

对于给定的样本集 D 和高斯混合成分的数目 K，先选取选取彼此距离尽可能大的向量作为初始的均值向量中心来初始化参数$\{\alpha_i, \mu_i, \Sigma_i \mid i \in \{1,2,\dots,k\}\}$和所有簇$C_\mathbf{i} = \emptyset, i = 1,2,\dots,k$

然后开始迭代直到迭代达到最大迭代次数或者参数值不再发生变化时退出迭代。

迭代过程为：

（1）E 步：根据式(4)计算每个样本由各个混合高斯成分生成的后验概率

（2）M 步：根据公式(8)(9)(14)来更新参数$\{\alpha_i, \mu_i, \Sigma_i \mid i \in \{1,2,\dots,k\}\}$

退出迭代后，根据式(5)来确定每个样本的簇标记λ_j，并将其加入对应的簇$\mathbf{C}_{\lambda_\mathbf{j}} = \mathbf{C}_{\lambda_\mathbf{j}} \cup \{\mathbf{x_j}\}$

最后输出簇划分$C = \{\mathbf{C}_1, \mathbf{C}_2, \dots, \mathbf{C_k}\}$

具体以注释形式给出。

```
def GMM_algorithm(self):
    print("GMM 算法：\n")
    for i in range(self.iteration_times):  # 进行迭代以得到最终的均值向量参数，协方差参数和混合系数
        print('第', i+1, "次迭代：")
        self.EM_algorithm()
        loss = np.linalg.norm(self.last_alpha - self.alpha) \
               + np.linalg.norm(self.last_mu - self.mu_data) \
               + np.sum([np.linalg.norm(self.last_sigma[i] - self.my_sigma[i]) for i in range(self.k)])  # 两次迭代的误差
        if loss > self.error:  # 如果新产生的均值向量参数，协方差参数和混合系数与原均值向量参数，协方差参数和混合系数的误差大于给定误差范围，则进行更新
            self.last_sigma = self.my_sigma
            self.last_mu = self.mu_data
            self.last_alpha = self.alpha
        else:  # 迭代终止条件：新产生的均值向量参数，协方差参数和混合系数与原均值向量参数，协方差参数和混合系数误差几乎可以忽略
            break
```

```
    self.EM_algorithm()  # 由于之只更新了均值向量参数，协方差参数和混合系数，需要再运行一次 EM 来得到新的似然值
    return self.mu_data, self.c


'''
EM 算法
'''
def EM_algorithm(self):
    temp_likelihoods = np.zeros((self.data_rows, self.k))  # 生成行为 data_rows，列为 k 的矩阵，作为后验概率公式的分母的一部分
    for i in range(self.k):  # 对每一列
        temp_likelihoods[:, i] = multivariate_normal.pdf(self.data, self.mu_data[i],
                                                         self.my_sigma[i])  # 得到所有数据在 mu_data[i]取值点附近的可能性
        # print("sadasdasdad",likelihoods[:, i])
    # 求期望 E，即 EM 算法的 E 步
    the_weighted_likelihoods = temp_likelihoods * self.alpha  # 还未求和的后验概率的分母
    sum_likelihoods = np.expand_dims(np.sum(the_weighted_likelihoods, axis=1), axis=1)  # 对后验概率的分母求和
    print('似然值为: ', np.log(np.prod(sum_likelihoods)))  # 输出似然值
    self.gamma = the_weighted_likelihoods / sum_likelihoods  # 根据公式，得到后验概率的值，即所有 gamma 的值
    # print('sadadawd',self.lamda)
    self.lamda = self.gamma.argmax(axis=1)  # 求得最大的 gamma 所在的簇标记 lamda，即得到了每个样本的簇标记
    for i in range(self.data_rows):  # 根据簇标记将所有的数据划分簇
        self.c[self.lamda[i]].append(self.data[i].tolist())
    # 最大化 M，即 EM 算法的 M 步
    for i in range(self.k):
        gamma = np.expand_dims(self.gamma[:, i], axis=1)  # 提取每一列,作为列向量
        self.mu_data[i] = (gamma * self.data).sum(axis=0) / gamma.sum()  # 根据公式，更新新的均值向量参数
        self.my_sigma[i] = (self.data - self.mu_data[i]).T.dot(
            (self.data - self.mu_data[i]) * gamma) / gamma.sum()  # 根据公式，更新新的协方差参数
        self.alpha = self.gamma.sum(axis=0) / self.data_rows  # 更新新的混合系数
```

```
'''
初始化高斯混合分布的模型参数中的 u 和 sigma
进行准备工作，生成初始簇中心点集合{u1，u2,...,uk},储存了所有的均值向量点，和生成
大小为 k×k 的对角矩阵 sigma
'''
def initial_params(self):
    # mu = np.array(self.data[random.sample(range(self.data_rows),
self.k)])
    # 随机选择 k 个点作为初始点 极易陷入局部最小值
    mu_collection = self.initial_cluster_center_point_by_maxlength()
# 选择彼此距离尽可能远的 K 个点作为初始簇中心点集合
    sigma = collections.defaultdict(list)  # 定义一个集合 sigma
    for i in range(self.k):  # 构建 k 个对角线元素为 0.1 的，对角矩阵作为
sigma 的值，是协方差矩阵
        sigma[i] = np.eye(self.data_columns, dtype=float) * 0.1
    return mu_collection, sigma


'''
选择彼此距离尽可能远的 K 个点作为初始簇中心点集合{u1，u2,...,uk},储存了所有的均值
向量点
'''
def initial_cluster_center_point_by_maxlength(self):
    mu_0 = np.random.randint(0, self.k) + 1  # 首先在 k 个点中随机选第一个
初始点
    mu_collection = [self.data[mu_0]]  # 定义一个初始簇中心点集合（k 集
合），将选取的第一个点放入其中
    for m in range(self.k - 1):  # 除了第一个点还需要选取 k-1 个点，每次选择
一个距离最大的点加入到 k 集合中
        all_length = []
        for i in range(self.data_rows):  # 对于样本集中的所有数据，求得其到
k 集合的距离
            all_length.append(np.sum(
                [self.distance_by_euclidean(self.data[i],
mu_collection[j]) for j in range(len(mu_collection))]))
        mu_collection.append(self.data[np.argmax(all_length)])  # 取距离
最大的点下标加入 k 集合
    print('初始均值向量集合为: \n', np.array(mu_collection))
    return np.array(mu_collection)
```

2.2.4 创建二维数据集

作用：

```
'''
生成 2 维数据集
参数中：
sample_means 表示 k 类数据的均值，以 list 的形式给出，如[[1, 2],[-1, -2], [0,
0]]
sample_number 表示 k 类数据的数量，以 list 的形式给出，如[10, 20, 30]
category_K 表示数据一共分为 k 类
'''
```

具体实现：给定参数生成数据即可

```
def create_data_two_dimensional(sample_means, sample_number,
category_K):
    covariance = [[0.1, 0], [0, 0.1]]  # 协方差
    sample_data = []
    for index in range(category_K):
        for times in range(sample_number[index]):
            sample_data.append(np.random.multivariate_normal(
                [sample_means[index][0], sample_means[index][1]],
covariance).tolist())
    return np.array(sample_data)
```

2.2.5 绘制图像

作用：

```
'''
画随机选择均值向量的 kmeans 图像
'''
'''
画彼此距离尽可能远的 K 个点的均值向量的 kmeans 图像
'''
'''
画彼此距离尽可能远的 K 个点的均值向量的 GMM 图像
'''
```

具体实现：调用绘制图像的函数即可。

```
def draw_k_means_by_random_selection(k, mu_random, c_random):
    for i in range(k):
        plt.scatter(np.array(c_random[i])[:, 0],
np.array(c_random[i])[:, 1], marker="x", label=str(i + 1))
    plt.scatter(mu_random[:, 0], mu_random[:, 1], facecolor="none",
edgecolor="r", label="center")
```

```
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title("k_means_by_random_selection")
    plt.show()

def draw_k_means_by_maxlength(k, mu_maxlength, c_maxlength):
    for i in range(k):
        plt.scatter(np.array(c_maxlength[i])[:, 0],
np.array(c_maxlength[i])[:, 1], marker="x", label=str(i + 1))
    plt.scatter(mu_maxlength[:, 0], mu_maxlength[:, 1],
facecolor="none", edgecolor="r", label="center")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title("k_means_by_maxlength")
    plt.show()

def draw_GMM(k, mu_gmm, c_gmm):
    for i in range(k):
        plt.scatter(np.array(c_gmm[i])[:, 0], np.array(c_gmm[i])[:,
1], marker="x", label=str(i + 1))
    plt.scatter(mu_gmm[:, 0], mu_gmm[:, 1], facecolor="none",
edgecolor="r", label="center")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title("GMM_by_maxlength")
    plt.show()
```

2.2.6 读取 iris.csv 数据集进行实验，并测试聚类的正确率

作用：

读取 iris.csv 文件即可。

具体实现：

```
class read_from_csv(object):
    def __init__(self):
        self.all_data = pd.read_csv("./iris.csv")  # 读取文件数据集
        self.data_x = self.all_data.drop('class', axis=1)  # 删除 class
```

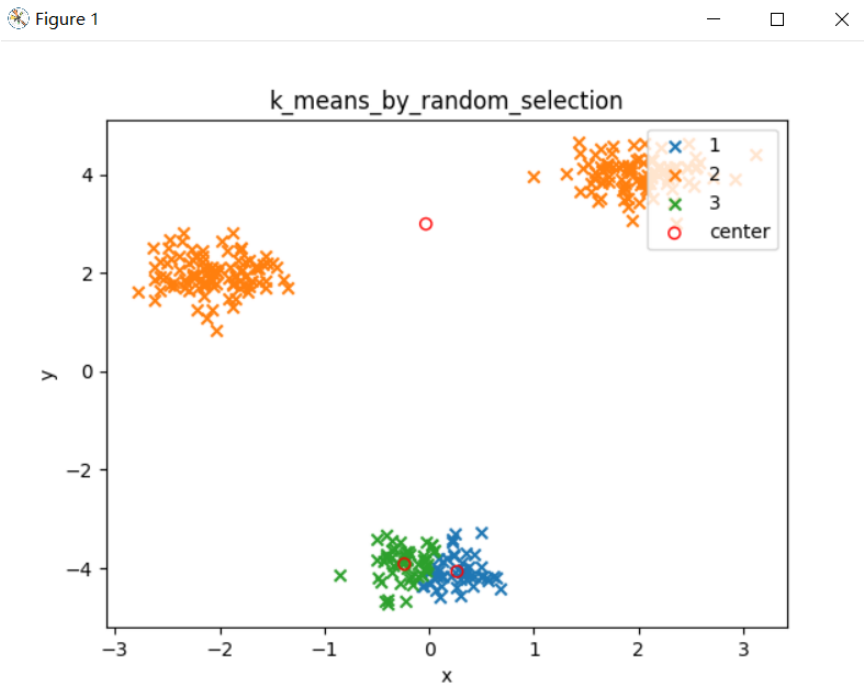
```
类别列作为数据集 data_x
        self.data_y = self.all_data['class']  # 将 class 类别列作为分类数据
集 data_y
        self.classes = list(itertools.permutations(['Iris-setosa',
'Iris-versicolor', 'Iris-virginica'], 3))

    '''
    返回拆分得到的数据集 data_x
    '''
    def get_data(self):
        return np.array(self.data_x, dtype=float)

    '''
    测试聚类的正确率
    '''
    def test_accuracy(self, y_label):
        number = len(self.data_y)
        counts = []
        for i in range(len(self.classes)):
            count = 0
            for j in range(number):
                if self.data_y[j] == self.classes[i][y_label[j]]:
                    count += 1
            counts.append(count)
        return np.max(counts) * 1.0 / number
```

3 实验结果分析

3.1 生成二维数据集，K-Means 两种初始均值向量结果对比

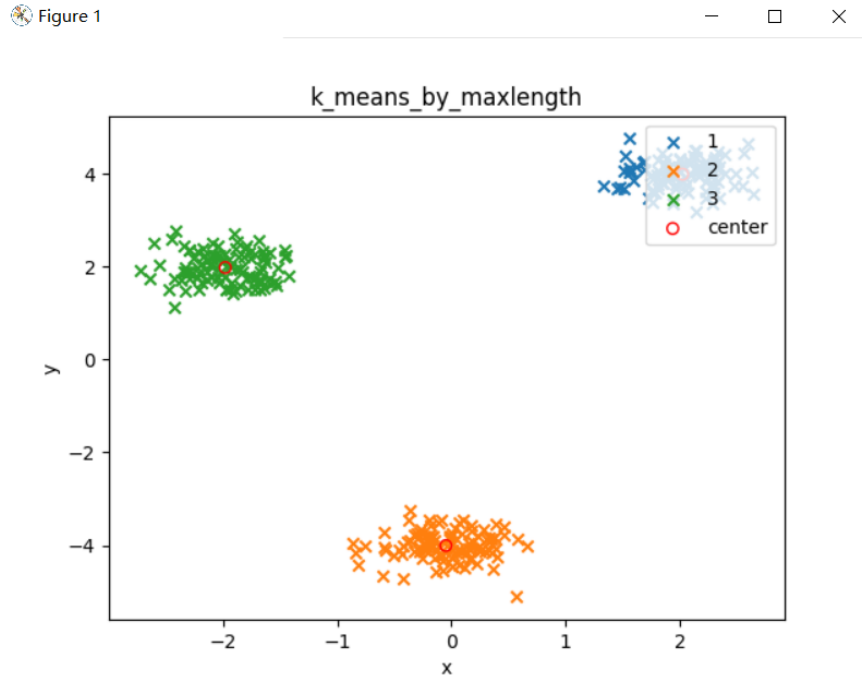
生成的数据集 k=3，均值分别[2, 4], [0, -4], [-2, 2]，三种数据数量分别为 100，一共 300 个数据，三个子数据集分布比较分散。

（1）K-Means 随机选取初值均值向量中心：



经过多次试验后，发现可能会出现糟糕的分类的结果，这是因为随机选择的三个初始均值向量距离太近导致陷入了局部最优解，没能有效将样本划分为三类。

（2）K-Means 选取彼此距离尽可能大的向量作为初始的均值向量中心：



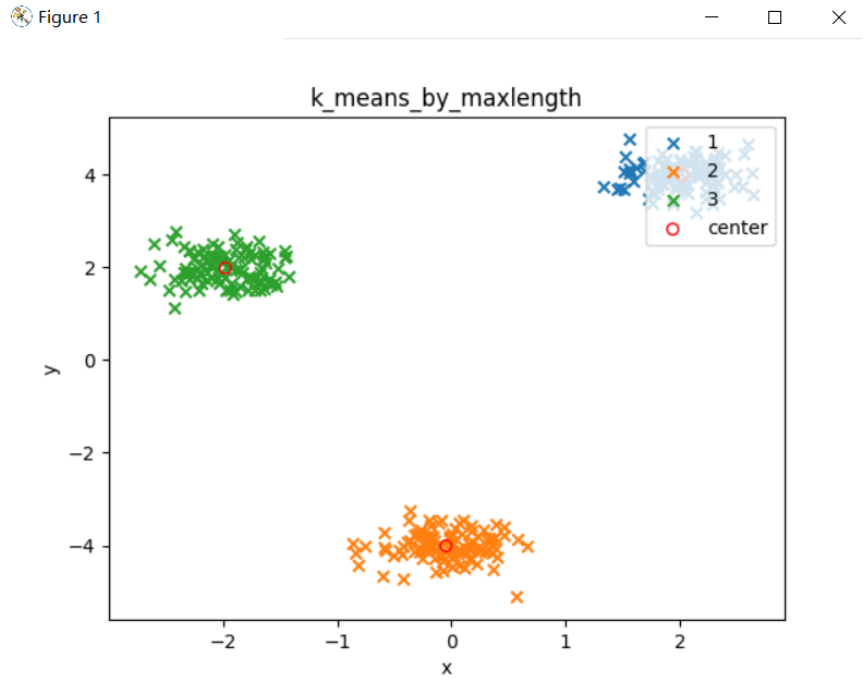
由于选择的初始均值向量中心距离很大，所以不会出现糟糕的分类结果。

3.2 生成二维数据集，K-Means 与 GMM 对比

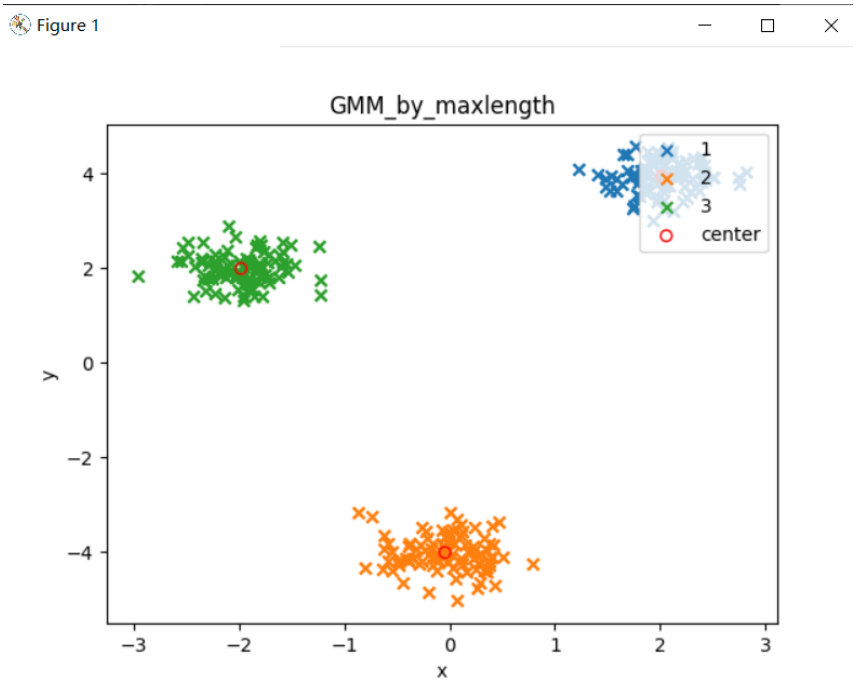
生成的数据集 k=3，均值分别[2, 4], [0, -4], [-2, 2]，三种数据数量分别为 100，一共 300 个

数据，三个子数据集分布比较分散。都选择彼此距离尽可能大的向量作为初始的均值向量中心

（1）K-Means 结果：



（2）GMM 结果：



通过对比可以看出，K-Means 和 GMM 两种方法进行聚类对表现出色。

3.3 测试 iris.csv 数据集

（1）GMM 算法在迭代中的似然值变化：

```
第 1 次迭代：
似然值为： -inf
第 2 次迭代：
似然值为： -274.5837689117663
第 3 次迭代：
似然值为： -255.018709331248
第 4 次迭代：
似然值为： -238.11192356345794
第 5 次迭代：
似然值为： -225.23891479668274
第 6 次迭代：
似然值为： -219.22643139995495
第 7 次迭代：
似然值为： -216.59866117954235
第 8 次迭代：
似然值为： -215.2584829114598
第 9 次迭代：
似然值为： -214.44663946001438
```

```
第 86 次迭代：
似然值为： -197.12593875493184
第 87 次迭代：
似然值为： -197.12593875493184
第 88 次迭代：
似然值为： -197.12593875493184
第 89 次迭代：
似然值为： -197.12593875493184
第 90 次迭代：
似然值为： -197.12593875493172
第 91 次迭代：
似然值为： -197.12593875493175
第 92 次迭代：
似然值为： -197.12593875493184
似然值为： -197.1259387549317

Process finished with exit code 0
```

似然值一直在增大且慢慢保持近乎不变，与预期相符。

（2）K-Means 与 GMM 的准确率：

与实际测试样本集的类别编号进行对比得到两种方法的准确率：

```
选取最远初始均值向量GMM的准确率为： 0.7133333333333334
选取最远初始均值向量的k-means的准确率为： 0.8866666666666667
```

4 结论

（1）K-Means 和 GMM 都是 EM 算法的体现，两者的共同之处是都有隐变量，并且都遵循 EM 算法的 E 步和 M 步的迭代优化。

（2）K-Means 和 GMM 的区别在于：K-Means 给出了很多很强的假设，比如假设数据式呈球状分布，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 使用更加一般的数据表示即高斯分布来描述聚类结果。

（3）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，而 GMM 高斯分布生成的后验概率来衡量。

（4）K-Means 的初始均值向量的选取对于最终的结果有很大的影响，如果选择不好的初始

均值向量（距离很近的初始均值向量）容易导致陷入局部最优解，可以通过选择彼此距离尽可能大的向量作为初始的均值向量

5 源代码

Lab3_1180300829.py

```
from Lab3 import k_means, GMM_EM, read_from_document
from Lab3.Create_data import create_data_two_dimensional
from Lab3.draw import draw_k_means_by_random_selection, draw_k_means_by_maxl
ength, draw_GMM

k = 3
means = [[2, 4], [0, -4], [-2, 2]]
number = [100, 100, 100]
data = create_data_two_dimensional(means, number, k)


kmeans = k_means.KMeans(data, k)
mu_random, c_random = kmeans.k_means_by_random_selection()
mu_maxlength, c_maxlength = kmeans.k_means_by_maxlength()


draw_k_means_by_random_selection(k, mu_random, c_random)

draw_k_means_by_maxlength(k, mu_maxlength, c_maxlength)

GMM = GMM_EM.GMM_EM_model(data, k)
mu_GMM, c_GMM = GMM.GMM_algorithm()

draw_GMM(k, mu_GMM, c_GMM)


iris = read_from_document.read_from_csv()
iris_data = iris.get_data()
GMM_iris = GMM_EM.GMM_EM_model(iris_data, k)
mu_iris, c_iris = GMM_iris.GMM_algorithm()


kmeans_iris = k_means.KMeans(iris_data, 3)
kmeans_mu_iris, kmeans_c_iris = kmeans_iris.k_means_by_maxlength()

```

```
print('选取最远初始均值向量 GMM 的准确率为：', iris.test_accuracy(GMM_iris.lamda))
print('选取最远初始均值向量的 k-means 的准确率为：', iris.test_accuracy(kmeans_iris.sample_assignments))
```

Create_data.py

```
import numpy as np


'''''
生成 2 维数据集
参数中：
sample_means 表示 k 类数据的均值，以 list 的形式给出，如[[1, 2],[-1, -2], [0, 0]]
sample_number 表示 k 类数据的数量，以 list 的形式给出，如[10, 20, 30]
category_K 表示数据一共分为 k 类
'''
def create_data_two_dimensional(sample_means, sample_number, category_K):
    covariance = [[0.1, 0], [0, 0.1]]  # 协方差
    sample_data = []
    for index in range(category_K):
        for times in range(sample_number[index]):
            sample_data.append(np.random.multivariate_normal(
                [sample_means[index][0], sample_means[index][1]], covariance).tolist())
    return np.array(sample_data)
```

draw.py

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

'''''
画随机选择均值向量的 kmeans 图像
'''
def draw_k_means_by_random_selection(k, mu_random, c_random):
    for i in range(k):
        plt.scatter(np.array(c_random[i])[:, 0], np.array(c_random[i])[:, 1], marker="x", label=str(i + 1))
    plt.scatter(mu_random[:, 0], mu_random[:, 1], facecolor="none", edgecolor="r", label="center")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
```

```
    plt.title("k_means_by_random_selection")
    plt.show()


'''''
画彼此距离尽可能远的 K 个点的均值向量的 kmeans 图像
'''
def draw_k_means_by_maxlength(k, mu_maxlength, c_maxlength):
    for i in range(k):
        plt.scatter(np.array(c_maxlength[i])[:, 0], np.array(c_maxlength[i])[:, 1], marker="x", label=str(i + 1))
    plt.scatter(mu_maxlength[:, 0], mu_maxlength[:, 1], facecolor="none", edgecolor="r", label="center")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title("k_means_by_maxlength")
    plt.show()


'''''
画彼此距离尽可能远的 K 个点的均值向量的 GMM 图像
'''
def draw_GMM(k, mu_gmm, c_gmm):
    for i in range(k):
        plt.scatter(np.array(c_gmm[i])[:, 0], np.array(c_gmm[i])[:, 1], marker="x", label=str(i + 1))
    plt.scatter(mu_gmm[:, 0], mu_gmm[:, 1], facecolor="none", edgecolor="r", label="center")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title("GMM_by_maxlength")
    plt.show()
```

GMM_EM.py

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


class GMM_EM_model(object):

```

```
8.      def __init__(self, data, k, error=1e-12, iteration_times=1000):
9.          self.data = data  # 数据集
10.         self.k = k  # k-means 的 k 值
11.         self.error = error  # 判断两个浮点数是否相等的误差值
12.         self.iteration_times = iteration_times  # 最大迭代次数
13.         self.data_rows, self.data_columns = self.data.shape
14.         self.alpha = np.ones(self.k) * (1.0 / self.k)
15.         self.mu_data, self.my_sigma = self.initial_params()  # 初始化初始簇中
    心点集合
16.         self.lamda = None  # 每个样本的簇标记
17.         self.c = collections.defaultdict(list)  # 簇划分
18.         self.last_alpha = self.alpha  # 保存更新后的混合系数
19.         self.last_mu = self.mu_data  # 保存更新后的均值向量系数
20.         self.last_sigma = self.my_sigma  # 保存更新后的协方差参数
21.         self.gamma = None  # 后验概率分布
22.
23.     @staticmethod
24.     def distance_by_euclidean(x1, x2):
25.         return np.linalg.norm(x1 - x2)
26.
27.     '''''
28.     选择彼此距离尽可能远的 K 个点作为初始簇中心点集合{u1，u2,...,uk},储存了所有的均
    值向量点
29.     '''
30.     def initial_cluster_center_point_by_maxlength(self):
31.         mu_0 = np.random.randint(0, self.k) + 1  # 首先在 k 个点中随机选第一个初
    始点
32.         mu_collection = [self.data[mu_0]]  # 定义一个初始簇中心点集合（k 集
    合），将选取的第一个点放入其中
33.         for m in range(self.k - 1):  # 除了第一个点还需要选取 k-1 个点，每次选择
    一个距离最大的点加入到 k 集合中
34.             all_length = []
35.             for i in range(self.data_rows):  # 对于样本集中的所有数据，求得其到
    k 集合的距离
36.                 all_length.append(np.sum(
37.                     [self.distance_by_euclidean(self.data[i], mu_collection[
    j]) for j in range(len(mu_collection))]))
38.             mu_collection.append(self.data[np.argmax(all_length)])  # 取距离
    最大的点下标加入 k 集合
39.         print('初始均值向量集合为：\n', np.array(mu_collection))
40.         return np.array(mu_collection)
41.
42.     '''''
43.     初始化高斯混合分布的模型参数中的 u 和 sigma
```

```
    进行准备工作，生成初始簇中心点集合{u1，u2,...,uk},储存了所有的均值向量点，和生成大小为 k×k 的对角矩阵 sigma
    '''
    def initial_params(self):
        # mu = np.array(self.data[random.sample(range(self.data_rows), self.k)])
        # 随机选择 k 个点作为初始点 极易陷入局部最小值
        mu_collection = self.initial_cluster_center_point_by_maxlength()  # 选择彼此距离尽可能远的 K 个点作为初始簇中心点集合
        sigma = collections.defaultdict(list)  # 定义一个集合 sigma
        for i in range(self.k):  # 构建 k 个对角线元素为 0.1 的，对角矩阵作为 sigma 的值，是协方差矩阵
            sigma[i] = np.eye(self.data_columns, dtype=float) * 0.1
        return mu_collection, sigma

    '''''
    EM 算法
    '''
    def EM_algorithm(self):
        temp_likelihoods = np.zeros((self.data_rows, self.k))  # 生成行为 data_rows，列为 k 的矩阵，作为后验概率公式的分母的一部分
        for i in range(self.k):  # 对每一列
            temp_likelihoods[:, i] = multivariate_normal.pdf(self.data, self.mu_data[i],
                                                             self.my_sigma[i])  # 得到所有数据在 mu_data[i]取值点附近的可能性
            # print("sadasdasdad",likelihoods[:, i])
        # 求期望 E，即 EM 算法的 E 步
        the_weighted_likelihoods = temp_likelihoods * self.alpha  # 还未求和的后验概率的分母
        sum_likelihoods = np.expand_dims(np.sum(the_weighted_likelihoods, axis=1), axis=1)  # 对后验概率的分母求和
        print('似然值为：', np.log(np.prod(sum_likelihoods)))  # 输出似然值
        self.gamma = the_weighted_likelihoods / sum_likelihoods  # 根据公式，得到后验概率的值，即所有 gamma 的值
        # print('sadadawd',self.lamda)
        self.lamda = self.gamma.argmax(axis=1)  # 求得最大的 gamma 所在的簇标记 lamda，即得到了每个样本的簇标记
        for i in range(self.data_rows):  # 根据簇标记将所有的数据划分簇
            self.c[self.lamda[i]].append(self.data[i].tolist())
        # 最大化 M，即 EM 算法的 M 步
        for i in range(self.k):
            gamma = np.expand_dims(self.gamma[:, i], axis=1)  # 提取每一列,作为列向量
```

```
            self.mu_data[i] = (gamma * self.data).sum(axis=0) / gamma.sum()  # 根据公式，更新新的均值向量参数
            self.my_sigma[i] = (self.data - self.mu_data[i]).T.dot(
                (self.data - self.mu_data[i]) * gamma) / gamma.sum()  # 根据公式，更新新的协方差参数
            self.alpha = self.gamma.sum(axis=0) / self.data_rows  # 更新新的混合系数

    '''''
    GMM 算法，实现对样本的聚类
    '''
    def GMM_algorithm(self):
        print("GMM 算法：\n")
        for i in range(self.iteration_times):  # 进行迭代以得到最终的均值向量参数，协方差参数和混合系数
            print('第', i+1, "次迭代：")
            self.EM_algorithm()
            loss = np.linalg.norm(self.last_alpha - self.alpha) \
                   + np.linalg.norm(self.last_mu - self.mu_data) \
                   + np.sum([np.linalg.norm(self.last_sigma[i] - self.my_sigma[i]) for i in range(self.k)])  # 两次迭代的误差
            if loss > self.error:  # 如果新产生的均值向量参数，协方差参数和混合系数与原均值向量参数，协方差参数和混合系数的误差大于给定误差范围，则进行更新
                self.last_sigma = self.my_sigma
                self.last_mu = self.mu_data
                self.last_alpha = self.alpha
            else:  # 迭代终止条件：新产生的均值向量参数，协方差参数和混合系数与原均值向量参数，协方差参数和混合系数误差几乎可以忽略
                break
        self.EM_algorithm()  # 由于之只更新了均值向量参数，协方差参数和混合系数，需要再运行一次 EM 来得到新的似然值
        return self.mu_data, self.c
```

k_means.py

```
import numpy as np
import collections
import random


class KMeans(object):
    def __init__(self, data, k, error=1e-6):
        self.data = data  # 数据集
        self.k = k  # k-means 的 k 值
```

```
        self.error = error  # 判断两个浮点数是否相等的误差值
        self.data_rows, self.data_columns = data.shape
        self.mu_datas = self.initial_cluster_center_point_by_maxlength()  # 初始簇中心点集合,即初始均值向量集合
        self.sample_assignments = [-1] * self.data_rows

    @staticmethod
    def distance_by_euclidean(x1, x2):
        return np.linalg.norm(x1 - x2)

    '''''
    选择彼此距离尽可能远的 K 个点作为初始簇中心点集合{u1，u2,...,uk},储存了所有的均值向量点
    '''
    def initial_cluster_center_point_by_maxlength(self):
        mu_0 = np.random.randint(0, self.k) + 1  # 首先在 k 个点中随机选第一个初始点
        mu_collection = [self.data[mu_0]]  # 定义一个初始簇中心点集合（k 集合），将选取的第一个点放入其中
        for m in range(self.k - 1):  # 除了第一个点还需要选取 k-1 个点，每次选择一个距离最大的点加入到 k 集合中
            all_length = []
            for i in range(self.data_rows):  # 对于样本集中的所有数据，求得其到 k 集合的距离
                all_length.append(np.sum([self.distance_by_euclidean(self.data[i], mu_collection[j])for j in range(len(mu_collection))]))
            mu_collection.append(self.data[np.argmax(all_length)])  # 取距离最大的点下标加入 k 集合
        print('初始均值向量集合为：\n', np.array(mu_collection))
        return np.array(mu_collection)

    '''''
    k_means 算法的实现，实现对样本的聚类
    '''
    def the_method_of_kmeans(self):
        number_of_times = 0  # 循环次数
        flag = 0  # 设置循环结束标签
        while True:
            c = collections.defaultdict(list)  # 初始化 Ci=空，i=1,2,3,...,k，C 为最终的初始均值向量集
            for i in range(self.data_rows):  # 对数据集中所有的点
                dij = [self.distance_by_euclidean(self.data[i], self.mu_datas[j]) for j in
```

```
                        range(self.k)]  # 对初始簇中心点集合中的所有均值向量点u1，u2,...,uk,求得两者的||xi-uj||
                lambda_j = np.argmin(dij)  # 求得数据集中最小的距离的簇的下标
                c[lambda_j].append(self.data[i].tolist())  # 将第 i 个点划分到C-lambda-j 对应的簇中
                self.sample_assignments[i] = lambda_j
            new_mu = np.array([np.mean(c[i], axis=0).tolist() for i in range(self.k)])  # 求解每个簇的均值作为新的均值向量
            flag = 0  # 初始化标签为 0
            for m in range(self.k):  # 对于所有的 i 属于 1,2,3,...,k，需要所有新的 ui 都等于旧的 ui,由于是浮点数，所以只需要两次的 ui 误差小于给定误差即可
                if self.distance_by_euclidean(self.mu_datas[m],
                                              new_mu[m]) > self.error:  # 若大于给定误差，则将新的 ui 赋值给旧的 ui，并置 flag 为 1
                    self.mu_datas[m] = new_mu[m]
                    flag = 1
            if flag == 0:  # 当没有新的 ui 产生后循环退出
                break
            print('kmeans 得到的均值向量的循环次数为:', number_of_times)
            number_of_times += 1
            print('得到的均值向量集合为: \n', self.mu_datas)
        return self.mu_datas, c

    '''''
    随机选择 k 个顶点作为初始簇中心点
    '''
    def k_means_by_random_selection(self):
        self.mu_datas = self.data[random.sample(range(self.data_rows), self.k)]  # 产生随机的 K 个点作为初始簇中心点集合{u1，u2,...,uk}
        return self.the_method_of_kmeans()

    '''''
    先随机选择第一个簇中心点 再选择彼此距离最大的 k 个顶点作为初始簇中心点
    '''
    def k_means_by_maxlength(self):
        self.mu_datas = self.initial_cluster_center_point_by_maxlength()  # 选择彼此距离尽可能远的 K 个点作为初始簇中心点集合{u1，u2,...,uk},
        return self.the_method_of_kmeans()
```

read_from_document.py

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

```
4.
5.
6.  class read_from_csv(object):
7.      def __init__(self):
8.          self.all_data = pd.read_csv("./iris.csv")  # 读取文件数据集
9.          self.data_x = self.all_data.drop('class', axis=1)  # 删除 class 类别列
    作为数据集 data_x
10.         self.data_y = self.all_data['class']  # 将 class 类别列作为分类数据集
    data_y
11.         self.classes = list(itertools.permutations(['Iris-setosa', 'Iris-
    versicolor', 'Iris-virginica'], 3))
12.
13.     '''''
14.     返回拆分得到的数据集 data_x
15.     '''
16.     def get_data(self):
17.         return np.array(self.data_x, dtype=float)
18.
19.     '''''
20.     测试聚类的正确率
21.     '''
22.     def test_accuracy(self, y_label):
23.         number = len(self.data_y)
24.         counts = []
25.         for i in range(len(self.classes)):
26.             count = 0
27.             for j in range(number):
28.                 if self.data_y[j] == self.classes[i][y_label[j]]:
29.                     count += 1
30.             counts.append(count)
31.         return np.max(counts) * 1.0 / number
```