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： 专业限选

实验题目： 实现 k-means 聚类方法

学号：7203610121
姓名： 刘天瑞

一、实验目的

实现一个 k-means 算法，并且用 EM 算法估计模型中的参数。

二、实验要求及实验环境

1. 实验要求

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），用 k-means 聚类，测试效果；

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. 实验环境

Windows 11； Visual Studio 2022 ； Python3.9.7 ：需要调用 numpy 库和 matplotlib 库

三、设计思想（本程序中的用到的主要算法及数据结构）

这一部分的实验中主要涉及设计的算法 K-means 算法在优化过程中本质上也是 EM 算法的具体应用，在 EM 算法的基础上加上了一些比较强的假设。因此，下面首先介绍 EM 算法的思想。

1. EM 算法原理

EM 算法是一种迭代优化策略算法，1977 年由 Dempster 等人总结提出，用于含有隐变量（Hidden variable）的概率模型参数的最大似然估计。

在 EM 算法中，它的计算方法中每一次迭代都分为两个步骤，其中一个为期望步（E 步），另一个为极大步（M 步）。每次迭代包含两个步骤：

1) E-step：求期望，调整分布；
2) M-step：根据 E 步调整的分布情况下，求使得优化目标函数取最大值时的参数，从而更新了参数

接着参数的更新又可以调整分布，不断循环往复，直到参数的变化收敛时训练结束，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。

EM 算法的具体流程图如下所示：

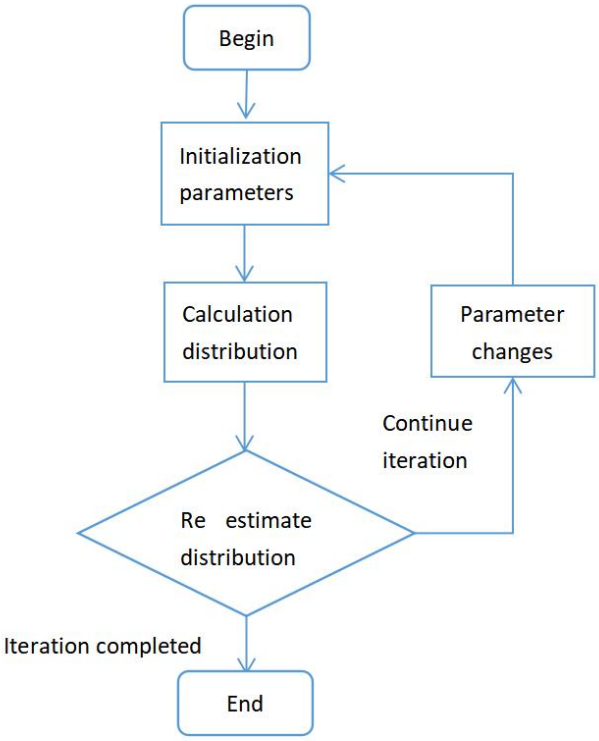


图 1 EM 算法的具体流程图

2. K-means 算法

K-means 算法主要解决的是无监督情况下的聚类问题，采取迭代优化的方法进行近似求解。

其解决问题的数学描述如下：假设给定训练样本集合为$X = \{x1, x2, \ldots, xn\}$，其中每一个训练样本都是一个 d 维的向量，向量的每一维表示样本的一个特征。假设需要将这一组样本集合聚类生成 k 个簇，也就是说给出 k 个标签，$C1, C2, \ldots, Ck$，给样本集中每一个样本打标签，达到的结果是使得聚类结果中每一个样本到样本中心的距离之和最小，因此我们的优化目标可以定为：

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

其中μi表示每一个簇的中心向量，表示方式为：$\mu_{\mathrm{i}} = \frac{1}{|\mathrm{C_i}|} \sum_{x \in C_i} x_i$ 其实也就是计算这一簇中所有向量的均值向量。则迭代优化过程如下：

1） 根据簇的数量 k 随机初始化 k 个向量作为每一个簇的初始中心向量，在本实验过程中随机选择样本集中的 k 个样本作为 k 个簇的中心样本；

2） 对于其余 n-k 个向量分别计算与这 k 个中心的距离，对于每一个向量都选择与其最近的一个中心，表示归于这个中心向量表示的簇中；（E

步）

3） 根据新的划分计算 k 个簇的中心向量，如果新的均值向量和旧的均值向量之间的差距已经达到精度要求，则表示迭代已经收敛，返回结果；如果还未收敛，返回第二步继续迭代。（M 步）

具体步骤也可以具体表示成如下所示：

1） 初始化 k 个聚类中心（k 为人为设定的常数）；

2） 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）。样本距离哪个类的聚类中心最近， 则将该样本划归到哪个聚类中心；

3） 根据划分结果重新计算每个类内真实的聚类中心，如果 k 个新算出聚类中心与前一次聚类中心的距离小于误差值，则停止迭代；否则，将此步中算出的聚类中心作为新的假设聚类中心，回到 1)继续迭代求解。（即循环直到聚类中心收敛）。

K-means 算法的大部分运行 Python 代码如下：

```
def kmeans(data, k, N, dim):
    "实现K-means算法"
    category = np.zeros((N, dim + 1))
    category[:, 0:dim] = data[:, :]#多出的一维矩阵用来保存类别标签
    center = np.zeros((k, dim))#k行dim列，用来保存各类中心
    for i in range(k):#随机以某些样本点作为初始点
        center[i, :] = data[np.random.randint(0, N), :]
    iter_count = 0#迭代次数
    #K-means算法的核心部分
    while True:
        iter_count += 1
        distance = np.zeros(k)#用来保存某次迭代中样本点到所有聚类中心的距离
        for i in range(N):
            point = data[i, :] #选取用来计算距离的样本点
            for j in range(k):
                t_center = center[j, :]#选取用来计算距离的聚类中心
                distance[j] = np.linalg.norm(point - t_center)#更新该样本点到聚类中心的距离
            min_index = np.argmin(distance)#找到该样本点对应距离最近的聚类中心
            category[i, dim] = min_index
        num = np.zeros(k)#保存每个类别的样本点数
        count = 0#计算更新后的距离小于误差值的聚类中心个数
        new_center_sum = np.zeros((k, dim))#临时变量
        new_center = np.zeros((k, dim))#保存此次迭代得到的新聚类中心
        for i in range(N):
            label = int(category[i, dim])
            num[label] += 1#统计各类别的样本点数
            new_center_sum[label, :] += category[i, :dim]
        for i in range(k):
            if num[i] != 0:
                new_center[i, :] = new_center_sum[i, :] / \
                    num[i]#计算本次样本点所得到的聚类中心
            new_k_distance = np.linalg.norm(new_center[i, :] - center[i, :])
            if new_k_distance < kmeans_epsilon:#计算更新后的距离小于误差值的聚类中心个数
                count += 1
        if count == k:#当所有聚类中心更新后的最小距离都小于误差值时，结束循环
            return category, center, iter_count, num
        else:#否则更新聚类中心
            center = new_center
```

图 2 K-means 算法的具体运行代码

3. 自行生成数据

在设计的实验中，需要用到高斯分布对算法实现进行测试，如果每次都逐个修改聚类数、方差以及均值等数据会非常麻烦，因此，可以直接将具体数据封装到 config 分布式配置类中。如下图所示，在 config 中封装了两个不同的聚类配置：

```
# 初始设置
config_0 = {
    'k': 4,#聚类数
    'n': 300,#每类的样本点数
    'dim': 2,#样本点维度
    'mu': np.array([[-5, 4], [5, 4], [3, -4], [-5, -5]]),#均值
    'sigma': np.array([[[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]]])#方差
}

config_1 = {
    'k': 8,#聚类数
    'n': 300,#每类的样本点数
    'dim': 2,#样本点维度
    'mu': np.array([[-4, 3], [4, 2], [1, -4], [-5, -3], [0, 0], [6, -1], [-1, 8], [7, -4]]),#均值
    'sigma': np.array([
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]],
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]]
    ])#方差
}

configs = [config_0, config_1]
```

图 3 将具体数据封装到 config 中

需要调用时，代码就会变得更为简洁且清晰：

```
#主函数
config = configs[con]#选取配置
k, n, dim, mu, sigma = config['k'], config['n'], config['dim'], config['mu'], config['sigma']
```

图 4 主函数部分调用 config 配置类

4. 计算准确率

考虑到经过算法更新后的 label 与初始生成时的 label 数值可能有所不同（例如，data 中的 label 是根据 0,1,2,3 的顺序分配的，而经过算法分出的 category 的标签可能第一组为 0 以外的其他数），因此，有必要采取其他方法计算分类准确率。

可以考虑在算法运行时统计出每个类别的样本点数，保存在 num 列表中，因为初始生成样本点时每组的样本点数量相同，因此可以尝试考虑用每组样本点的数量减去均值，再求其绝对值之和，最后除以二便是分类错误的样本点的数目，依此就可以较为简单地计算出分类的准确率了。

计算准确率的具体运行代码实现如下图所示：

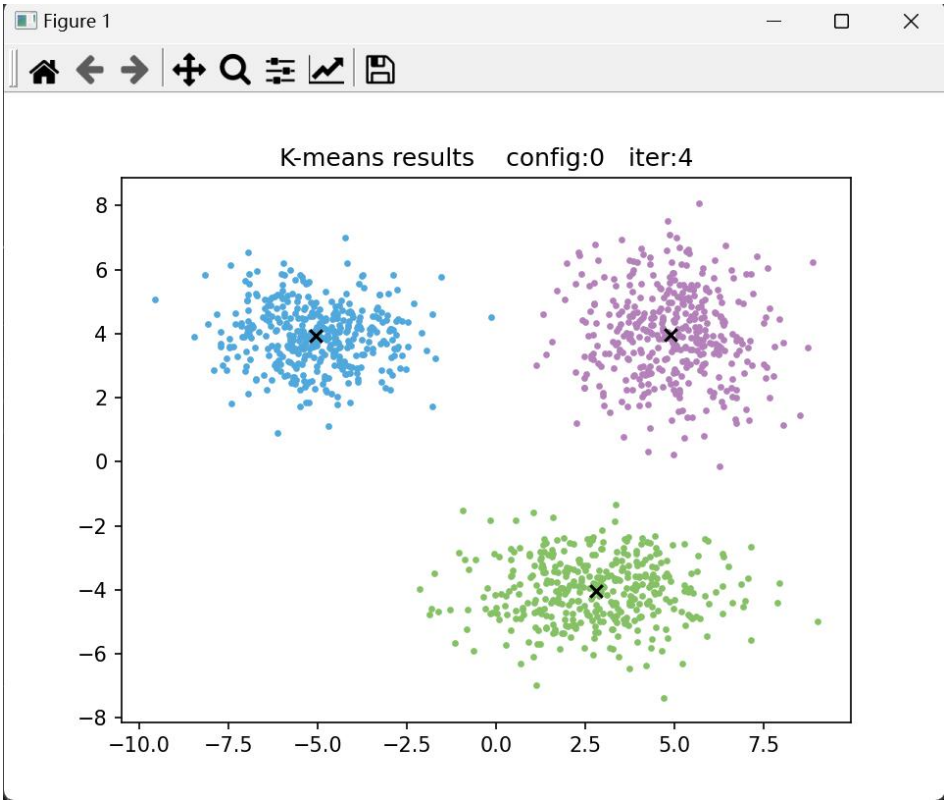
```
def calculate_accuracy(num, n, N):
    "计算K-means算法的分类准确率"
    """主要思想：首先将各类样本点数都减去最开始分类时每一类别的总数目，再求其中元素的绝对值之和，
    然后除以2，最后得到的即为分类错误点的个数"""
    num_1 = np.abs(num - n)
    error = np.sum(num_1) / 2
    return 1 - error / N
```

图 5 计算 K-means 算法准确率的具体运行代码

四、实验结果与分析

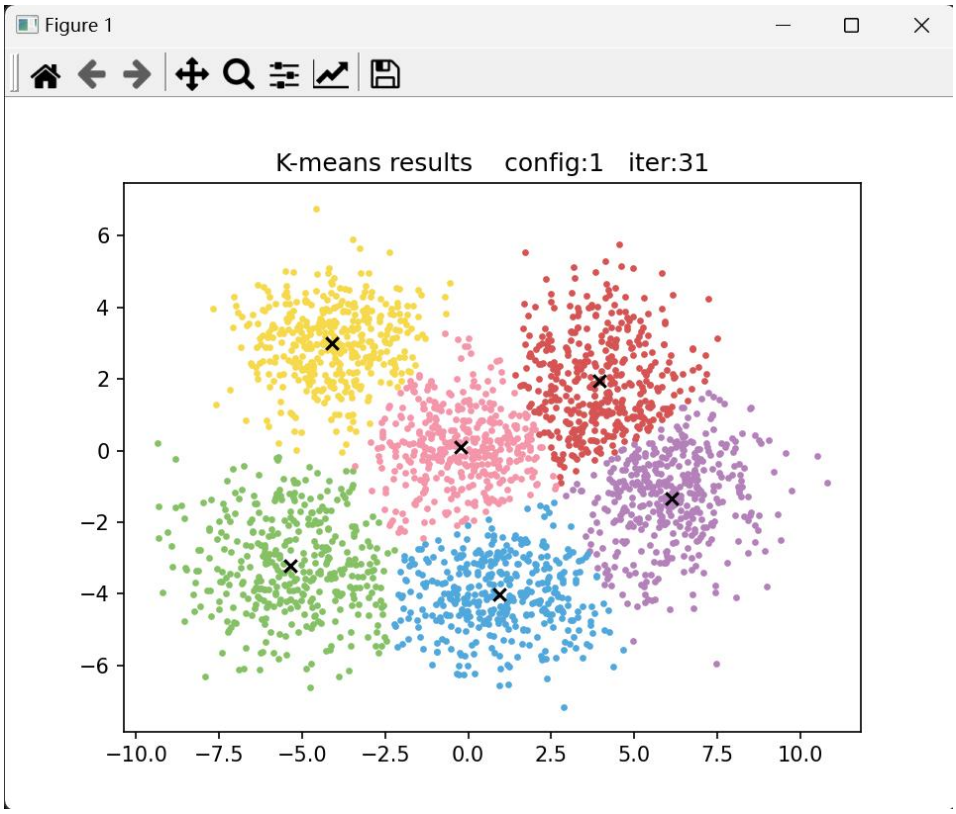
1.K-means 算法运行结果

运行 K-means 算法的程序，自行生成数据，不妨设置 K-means 算法的迭代误差 υ 为 1e-7，然后可以观察迭代过程分别在 config_0 与 config_1 配置下的运行效果，如下图所示：




```
accuracy:0.9991666666666666
```

图 6 在 config_0 配置下的运行结果




```
accuracy:0.9891666666666666
```

图 7 在 config_1 配置下的运行结果

从上图可以看出:在总共两种配置下 K-means 算法聚类模型均取得了十分不错的分类聚类效果，并且在实际运行时，运行速度也较快，时间复杂度较低。

2. 采用 UCI 数据集

本次实验我选取了 UCI 数据网站里的 Iris 数据集对算法进行了测试。从数学角度来看 Iris 数据集包含 150 个样本数据，并且每个样本数据由四个维度构成（即花萼长度，花萼宽度，花瓣长度，花瓣宽度）。其中需要根据这四个维度的数据来预测鸢尾花属于(Setosa, Versicolour, Virginica)三类中的哪一类。所以在样本中，每一类鸢尾花有 50 个样本。

尝试读取 Iris 数据集并进行测试，测试结果如下图所示：

```
K-means算法结果:
迭代次数: 4    分类准确率:0.67
```

```
K-means算法结果:
迭代次数: 5    分类准确率:0.83
```

```
K-means算法结果:
迭代次数: 6    分类准确率:0.92
```

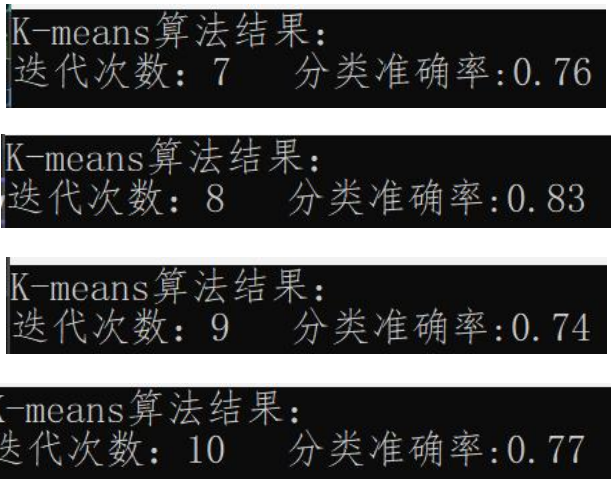


图 8 不同迭代次数的 K-means 算法结果

可以注意到，在 7 次实验中，K-means 算法总体准确率较高，但有一次的准确率低至 0.67，这可能是因为 K-means 算法的聚类结果严重依赖于初始化时的聚类中心，当初始化后的聚类中心分类效果不好（例如相对距离过近）时，会导致整体聚类的表现结果较差，因此可以尝试进行多次实验来选择最好的分类效果（择优）。

五、结论

1. K-means 算法可以说是 EM 算法的体现：它遵循 EM 算法的 E 步和 M 步的迭代优化模式。但是 K-means 算法的假设更强，相对来说也更好理解；
2. 根据实验结果，总体上来讲 K-means 算法能较好地解决简单的分类问题，算法速度相对来说更快，但其准确率无法反应模型的真实效果；
3. K-means 算法存在着可能只取到局部最优的问题，对于 K-means 算法来说，初值的选取对其结果影响较大，解决方法是考虑多次实验取比较平均的结果作为最终分类结果。
4. 事实上 K-means 就是 GMM 的一种极特殊情况假设每种类在样本中出现的概率相等均为 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，这样就可以使用欧式距离之和代替最大似然函数来优化。而且对于 K-means 来说，将 GMM 简化为每一个样本点完全属于一个高斯分布，在更新的过程中不考虑多个高斯分布混合生成的情况。而且 K-means 比较依赖于初始化，初始化的差异可能导致比较大的结果差异。由于做了这么多假设，而真实数据大概率上并不会满足这些假设，因此大多数情况下 K-means 的效果比 GMM 略差。

六、参考文献

[1] 周志华,《机器学习》,清华大学出版社，2016

[2] 阿泽.【机器学习】K-means（非常详细）[EB/OL].2020[2021-10-23]. https://zhuanlan.zhihu.com/p/78798251.

[3] UCI Machine Learning Repository.Iris DataSet[EB/OL].1988[2021-10-24].

https://archive.ics.uci.edu/ml/datasets/Iris.

七、附录：源代码（带注释）

```
 # -*- coding: gbk -*-

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
import pandas as pd

K_means_epsilon = 1e-7#K-means算法的迭代误差
condition = 1#选择的设置:config_0：0；config_1：1
method = 1#选择的方法：0：K-means；1：UCI
colors = ['#86C166', '#51A8DD', '#B481BB', '#F596AA', '#F7D94C', '#D75455']#提供颜色库

#初始化设置
config_0 = {
    'k': 3,#聚类数
    'n': 400,#每类的样本点数
    'dim': 2,#样本点的维度
    'mu': np.array([[-5, 4], [5, 4], [3, -4], [-5, -5]]),#均值
    'sigma': np.array([[[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0,
2]]])#方差
}

config_1 = {
    'k': 6,#聚类数
    'n': 400,#每类的样本点数
    'dim': 2,#样本点的维度
    'mu': np.array([[-4, 3], [4, 2], [1, -4], [-5, -3], [0, 0], [6, -1], [-1, 8], [7, -4]]),#
均值
    'sigma': np.array([
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]],
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]]
    ])#方差
}

configs = [config_0, config_1]

def generate_data(k, n, dim, mu, sigma):
    "生成数据"
    raw_data = np.zeros((k, n, dim))
    for i in range(k):
```

```
        raw_data[i] = np.random.multivariate_normal(mu[i], sigma[i], n)
    data = np.zeros((k * n, dim))
    for i in range(k):
        data[i * n:(i + 1) * n] = raw_data[i]
    return data


def K_means(data, k, N, dim):
    "实现K-means算法"
    category = np.zeros((N, dim + 1))
    category[:, 0:dim] = data[:, :]#多出的一维矩阵用来保存类别标签
    center = np.zeros((k, dim))#k行dim列，用来保存各类中心
    for i in range(k):#随机以某些样本点作为初始点
        center[i, :] = data[np.random.randint(0, N), :]
    iter_count = 0#迭代次数
    #K-means算法的核心部分
    while True:
        iter_count += 1
        distance = np.zeros(k)#用来保存某次迭代中样本点到所有聚类中心的距离
        for i in range(N):
            point = data[i, :] #选取用来计算距离的样本点
            for j in range(k):
                t_center = center[j, :]#选取用来计算距离的聚类中心
                distance[j] = np.linalg.norm(point - t_center)#更新该样本点到聚类中心的
距离
            min_index = np.argmin(distance)#找到该样本点对应距离最近的聚类中心
            category[i, dim] = min_index
        num = np.zeros(k)#保存每个类别的样本点数
        count = 0#计算更新后的距离小于误差值的聚类中心个数
        new_center_sum = np.zeros((k, dim))#临时变量
        new_center = np.zeros((k, dim))#保存此次迭代得到的新聚类中心
        for i in range(N):
            label = int(category[i, dim])
            num[label] += 1#统计各类别的样本点数
            new_center_sum[label, :] += category[i, :dim]
        for i in range(k):
            if num[i] != 0:
                new_center[i, :] = new_center_sum[i, :] / \
                    num[i]#计算本次样本点所得到的聚类中心
            new_k_distance = np.linalg.norm(new_center[i, :] - center[i, :])
            if new_k_distance < K_means_epsilon:#计算更新后的距离小于误差值的聚类中心个
数
                count += 1
        if count == k:#当所有聚类中心更新后的最小距离都小于误差值时，结束循环
```

```
            return category, center, iter_count, num
        else:#否则更新聚类中心
            center = new_center


def calculate_accuracy(num, n, N):
    "计算K-means算法的分类准确率"
    """主要思想：首先将各类样本点数都减去最开始分类时每一类别的总数目，再求其中元素的绝对值之和，
    然后除以2，最后得到的即为分类错误点的个数"""
    num_1 = np.abs(num - n)
    error = np.sum(num_1) / 2
    return 1 - error / N


def K_means_show(k, n, dim, mu, sigma):
    "K-means算法的结果展示"
    data = generate_data(k, n, dim, mu, sigma)
    #print(data.shape)#(1200,2)
    N = data.shape[0]#N为样本点总数
    category, center, iter_count, num = K_means(data, k, N, dim)
    accuracy = calculate_accuracy(num, n, N)
    for i in range(N):#绘制已分类的样本点
        color_num = int(category[i, dim] % len(colors))
        plt.scatter(category[i, 0], category[i, 1],
                    c=colors[color_num], s=5)
    for i in range(k):
        plt.scatter(center[i, 0], center[i, 1],
                    c='red', marker='x')#绘制所有的聚类中心
    print("accuracy:" + str(accuracy))
    plt.title("K-means results " + "   config:" +
              str(condition) + "   iter:" + str(iter_count))
    plt.show()
    return


def UCI_read():
    "读入UCI数据并进行切分"
    raw_data = pd.read_csv("./iris.csv")
    data = raw_data.values
    label = data[:, -1]
    np.delete(data, -1, axis = 1)
    #print(data.shape)#(150, 5)
    return data, label
```

```
def UCI_show():
    "使用UCI数据集测试K-means算法"
    data, _ = UCI_read()
    k = 6#聚类数
    N = data.shape[0]#样本数量
    n = N / k#每一类样本数量
    dim = data.shape[1]#样本维度
    #K-means算法的计算结果
    _, _, iter_count, num = K_means(data, k, N, dim)
    K_means_accuracy = calculate_accuracy(num, n, N)
    print("K-means算法结果：")
    print("迭代次数：%d    分类准确率:%.2f\n" % (iter_count, K_means_accuracy))
    return


#主函数
config = configs[condition]#选取配置
k, n, dim, mu, sigma = config['k'], config['n'], config['dim'], config['mu'],
config['sigma']
if method == 0:
    K_means_show(k, n, dim, mu, sigma)
elif method == 1:
    UCI_show()
```