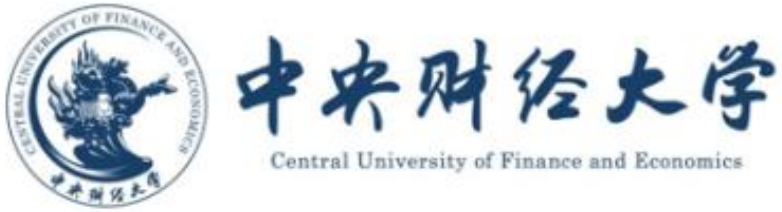


k-means 算法的改进与测试——基于矩阵运算的实现

学年学期：2019-2020 第二学期

课程名称：数据挖掘

课程代码：3320062

任课教师：马景义

姓　　名：徐菡

学　　号：2017310719

班　　级：金融实验班 17

摘　要

本文对聚类中的基本算法 k-means 算法进行了算法描述和 Python 实现，并对初始聚类中心选择、最优 k 值确定以及聚类中心计算方法进行了进一步讨论和实现，然后分别用模拟数据和真实数据类来对代码进行测试，均得到了较优的聚类结果。

关键词：k-means 算法　聚类 k 值　初始聚类中心　k-medoids 算法

ABSTRACT

This report describes the basic procedure and the Python implementation of k-means clustering algorithm, and discusses the determination of initial clustering centers, optimal clusters and the method of deciding the clustering centers. The codes prove to be effective on both simulated data and real data.

KEY WORDS:　K-means algorithm　Initial clustering centers　Number of clusters

目　　录

1. 算法实现——自定义 Kmeans 类

聚类分析是数据挖掘中的一种重要的分析方法，它的目标是将数据集合分成若干簇，使得同一簇内的数据点相似度尽可能大，而不同簇间的数据点相似度尽可能小。学界陆续提出了多种基于不同思想的聚类算法，主要有基于划分的算法、基于层次的算法、基于密度的算法、基于网格的算法和基于模型的算法等。这些算法都能取得不错的聚类效果，其中应用最多且算法思想较为简单的是基于划分的 k-means 算法。（吴夙慧等，2011）

1.1 初始聚类中心优化的 k-means 算法

1.1.1 一般 k-means 算法及其缺陷

一般的 k-means 算法描述如下：

输入：聚类个数 k 以及包含 n 个数据对象的数据集；

输出：满足目标函数值最小的 k 个聚类。

算法流程：

（1）从 n 个数据对象中任意选择 k 个对象作为初始聚类中心；

（2）循环下述流程(3)到(4)，直到目标函数 J 取值不再变化；

（3）根据每个聚类对象的均值(中心对象)，计算每个对象与这些中心对象的距离，并且根据最小距离重新对相应对象进行划分；

（4）重新计算每个聚类的均值（中心对象）。

传统的 k-means 算法的缺陷是对初始聚类中心敏感，不同的初始中心往往对应着不同的聚类结果。因此我们想找到一组能反映数据分布特征的数据对象作为初始聚类中心，也就是改进上述算法中的第(1)步。

1.1.2 优化初始聚类中心的 k-means 算法

为了避免取到噪声点，取相互距离最远的 k 个处于高密度区域的点作为初始聚类中心。为了计算数据对象x_i所处区域的密度，定义一个密度参数：以x_i为中心，包含常数 Minpts 个数据对象的半径称之为对象x_i的密度参数，用 ε 表示。ε

越大，说明数据对象所处区域的数据密度越低。反之，ε 越小，说明数据对象所处区域的数据密度越高。通过计算每个数据对象的密度参数，就可以发现处于高密度区域的点，从而得到一个高密度点集合 D。

优化初始聚类中心的 k-means 算法描述：

输入：聚类个数 k 以及包含 n 个数据对象的数据集；

输出：满足目标函数值最小的 k 个聚类。

（1）计算任意两个数据对象间的距离 d(x_i, x_j)；

（2）计算每个数据对象的密度参数，把处于低密度区域的点删除，得到处于高密度区域的数据对象的集合 D；

（3）把处于最高密度区域的数据对象作为第 1 个中心z_1；

（4）把z_1距离最远的数据对象作为第 2 个初始中心z_2，z_2∈D；

（5）令z_3为满足 $max(min(d(x_i,z_1),d(x_i,z_2)),i=1,2,\dots,n$ 的数据对象x_i，z_3∈D；

（6）令z_4为满足 $max(min(d(x_i,z_1),d(x_i,z_2),d(x_i,z_3)),i=1,2,\dots,n$ 的x_i，z_4∈D；

…

（7）令z_k为满足$max(min(d(x_i,z_j),d(x_i,z_2)),i=1,2,\dots,n,j=1,2,\dots,k-1$的$x_i$，$z_k$∈D；

（8）从这 k 个聚类中心出发，应用 k-means 聚类算法，得到聚类结果。

1.1.3 自定义 Kmeans 类介绍

（1）参数

minpts：用以 xi 为中心，包含常数 minpts 个数据对象的半径来衡量 xi 的密度

p：初始中心备选点的个数

k：聚类个数

center_method：计算聚类中心的方法，默认值为"mean"

（2）方法

init_density()：初始化每个对象的密度

init_centroid()：初始化中心

k_means()：执行 k-means 算法

plot_cluster()：在二维平面画出聚类结果

cal_BWP()：计算平均 BWP 指标（在 1.2 节中详述）

cal_all()：将上述方法一次性执行

new_assign()：对于新数据进行分类（将新数据划分到已经聚好的类中）

（3）属性

data：待聚类的数据

n：数据对象的个数

dim：数据维数，即特征个数（一般用 k 表示，但在本类中 k 表示聚类个数，因此用 dim 代替）

distance：对象两两之间的距离的矩阵，distance[i][j]表示第 i 和对象和第 j 个对象之间的距离

density：每个对象的密度

centroid：每个类的中心坐标，初始值为由上述方法选出的 k 个点

J：判断迭代是否结束的目标函数，若两次相邻迭代后 J 保持不变，则停止迭代

point_cluster_dist：每个对象到 k 个类中心的距离

point_cluster_assign：每个对象距离最近的中心在 centroid 数组中的索引

cluster：列表存放每个类中的对象在 data 中的索引

avg_BWP：当前参数下的平均 BWP 指标值，用于选择最优参数

自定义 Kmeans 类代码如下：

```
# Kmeans 类，用于执行上述算法步骤
class Kmeans(object):
    def __init__(self, data, minpts, p, k, center_method="mean"):
        self.data = data
        self.minpts = minpts  # 用以 xi 为中心，包含常数 minpts 个数据对象
```

```
的半径来衡量xi 的密度
        self.n = data.shape[0]  # 数据对象的个数
        self.dim = data.shape[1]  # 每个对象的维度
        self.p = p  # 取p 个密度较大的点作为初始中心的备选点
        self.k = k  # 聚成的类的个数
        self.center_method = center_method  # 计算中心的方法
        self.distance = mm_EuDistance(data, data)  # 对象两两之间的距离
的矩阵，distance[i][j]表示第i 和对象和第j 个对象之间的距离
        self.density = np.zeros(self.n)  # 每个对象的密度
        self.centroid = np.zeros([k, self.dim])  # 每个类的中心坐标，初
始值为由上述方法选出的k 个点
        self.J = 0  # 判断迭代是否结束的目标函数，若两次相邻迭代后J 保持不
变，则停止迭代
        self.point_cluster_dist = np.zeros([self.n, k])  # 每个对象到k
个类中心的距离
        self.point_cluster_assign = np.zeros(self.n)  # 每个对象距离最
近的中心在centroid 数组中的索引
        self.cluster = []  # cluster 列表存放每个类中的对象在data 中的索
引
        self.avg_BWP = 0  # 当前参数下的平均BWP 指标值，用于选择最优参数

    # 初始化每个对象的密度
    def init_density(self):
        for i in range(self.n):
            nearest = self.distance[i].argsort()[:self.minpts + 1]  #
每个对象距离最近的minpts 个点（包含对象自身）
            self.density[i] = self.distance[nearest][:, nearest].sum()
  # 这minpts 个对象的两两距离之和，density 越小，密度越大

    # 初始化中心
    def init_centroid(self):
        high = self.density.argsort()[:self.p]  # 取density 中前p 个对
象，即密度最大的p 个对象
        used = [high[0]]  # used 列表表示已经选出的作为初始中心的对象
        unused = high[1:]  # 剩下的没有作为初始中心的对象
        pick = [0] * self.k
        # 选剩下的k-1 个初始中心
        for i in range(self.k - 1):
            pick[i] = np.min(self.distance[used][:, unused])
            used.append(np.where(self.distance == np.max(pick[i]))
[0].tolist()[0])
            used.append(np.where(self.distance == np.max(pick[i]))
[0].tolist()[1])
            used = np.unique(used)
```

```
            used = used.tolist()
            unused = list(filter(lambda x: x not in used, high))
        self.centroid = self.data[used]

    def k_means(self):
        iteration = 0  # 记录迭代次数
        j_changed = True
        while j_changed:
            # 根据每个类对象的均值(中心对象)，计算每个对象与这些中心对象的
距离
            self.point_cluster_dist = mm_EuDistance(self.data, self.ce
ntroid)
            # 根据最小距离重新对相应对象进行划分
            self.point_cluster_assign = np.argmin(self.point_cluster_
dist, axis=1)
            # 重新计算每个类的均值（中心对象）
            # centroid[0] = cal_mean(np.where(point_cluster_assign==
0))
            self.centroid = np.array(
                list(map(lambda j: cal_center(self.data[np.where(self.
point_cluster_assign == j)], self.center_method),
                         list(range(self.k)))))
            # print(centroid)
            iteration += 1
            # 判断目标函数J是否改变，并更新J
            j_new = np.sum(self.point_cluster_dist)
            if self.J == j_new:
                j_changed = False
            self.J = j_new
        print("迭代次数为：")
        print(iteration)
        return self.centroid

    # 只能画出二维数据
    def plot_cluster(self):
        if self.dim != 2:
            print("对不起，本模块无法画出三维及以上数据!")
            return 1
        mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r',
'pr']
        if self.k > len(mark):
            print("对不起，本模块无法画出十一及以上类别")
            return 1
        # data_sort = self.data.append(self.point_cluster_assign, axi
```

```
s=1)
        for i in range(self.n):
            markIndex = int(self.point_cluster_assign[i])  # 为样本指定
颜色
            plt.plot(self.data[i][0], self.data[i][1], mark[markInde
x])
        mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b',
'pb']
        # 画出中心点
        for i in range(self.k):
            plt.plot(self.centroid[i][0], self.centroid[i][1], mark
[i], markersize=12)
        plt.show()
```

1.2 K-means 算法最佳聚类数确定方法

一般来说，一个好的聚类划分应尽可能反映数据集的内在结构，使类内样本尽可能相似，类间样本尽可能不相似。从距离测度考虑，就是使类内距离极小化而类间距离最大化的聚类是最优聚类。本报告采用了（周世兵等，2010）论文中设计的一种新的聚类有效性指标，该指标可以对 K-means 算法的聚类结果进行评估并可用来确定最佳聚类数。

1.2.1 类内距离和类间距离的定义和计算公式

定义 1 令 K = {X，R}为聚类空间，其中 X = $\{x_1, x_2, \ldots, x_n\}$，假设 n 个样本对象被聚类为 c 类，定义第 j 类的第 i 个样本的最小类间距离 b(j,i)为该样本到其他每个类中样本平均距离的最小值：

$$b(j,i) = \min_{1\le k\le c, k\ne j}\left(\frac{1}{n_k}\sum_{p=1}^{n_k} \| x_p^{(k)} - x_i^{(j)} \|^2\right)$$

其中: k 和 j 表示类标，$x_i^{(j)}$表示第 j 类的第 i 个样本，$x_p^{(k)}$表示第 k 类的第 p 个样本，n_k表示第 k 类中的样本个数，$\|.\|^2$表示平方欧氏距离。

定义 2 令 K = {X，R}为聚类空间，其中 X = $\{x_1, x_2, \ldots, x_n\}$，假设 n 个样本对象被聚类为 c 类，定义第 j 类的第 i 个样本的类内距离 w(j,i)为该样本到第 j 类中其他所有样本的平均距离，即:

$$w(j,i)=\frac{1}{n_j-1}\sum_{q=1,q\neq i}^{n_j}\|x_q^{(j)}-x_i^{(j)}\|^2$$

其中: $x_q^{(j)}$表示第 j 类中的第 q 个样本，并且 q≠i, n_j表示第 j 类中的样本个数。实际使用中，无需保证 q≠i，因为 q=i 时，欧氏距离为 0，并不影响算法的正确性。

定义 3 令 K = {X，R} 为聚类空间，其中 X = $\{x_1,x_2,\ldots,x_n\}$，假设 n 个样本对象被聚类为 c 类，定义第 j 类的第 i 个样本的聚类距离 baw(j,i)为该样本的最小类间距离和类内距离之和，即:

$$baw(j,i)=b(j,i)+w(j,i)$$
$$=\min_{1\le k\le c,k\neq j}(\frac{1}{n_k}\sum_{p=1}^{n_k}\|x_p^{(k)}-x_i^{(j)}\|^2)+\frac{1}{n_j-1}\sum_{q=1,q\neq i}^{n_j}\|x_q^{(j)}-x_i^{(j)}\|^2$$

定义 4 令 K = {X，R} 为聚类空间，其中 X = $\{x_1,x_2,\ldots,x_n\}$，假设 n 个样本对象被聚类为 c 类，定义第 j 类的第 i 个样本的聚类离差距离 bsw(j,i)为该样本的最小类间距离和类内距离之差，即:

$$bsw(j,i)=b(j,i)+w(j,i)$$
$$=\min_{1\le k\le c,k\neq j}(\frac{1}{n_k}\sum_{p=1}^{n_k}\|x_p^{(k)}-x_i^{(j)}\|^2)-\frac{1}{n_j-1}\sum_{q=1,q\neq i}^{n_j}\|x_q^{(j)}-x_i^{(j)}\|^2$$

定义 5 令 K = {X，R} 为聚类空间，其中 X = $\{x_1,x_2,\ldots,x_n\}$，假设 n 个样本对象被聚类为 c 类，定义第 j 类的第 i 个样本的类间类内划分(Between-Within Proportion，BWP)指标 BWP(j,i)为该样本的聚类离差距离和聚类距离的比值，即:

$$BWP=\frac{\min_{1\le k\le c,k\neq j}(\frac{1}{n_k}\sum_{p=1}^{n_k}\|x_p^{(k)}-x_i^{(j)}\|^2)-\frac{1}{n_j-1}\sum_{q=1,q\neq i}^{n_j}\|x_q^{(j)}-x_i^{(j)}\|^2}{\min_{1\le k\le c,k\neq j}(\frac{1}{n_k}\sum_{p=1}^{n_k}\|x_p^{(k)}-x_i^{(j)}\|^2)+\frac{1}{n_j-1}\sum_{q=1,q\neq i}^{n_j}\|x_q^{(j)}-x_i^{(j)}\|^2}$$

BWP 指标反映了单个样本的聚类有效性情况，BWP 指标值越大，说明单个样本的聚类效果越好。我们通过求某个数据集中所有样本的 BWP 指标值的平均值，来分析该数据集的聚类效果。显然，平均值越大，说明该数据集的聚类效果越好，其最大值所对应的聚类数是最佳聚类数。由此我们得到如下公式，其中

$avg_BWP(k)$表示数据集聚成 k 类时的平均 BWP 指标值，k_opt表示最佳聚类数。

$$avg_{BWP}(k) = \frac{1}{n}\sum_{j=1}^{k}\sum_{i=1}^{n_j} BWP(j,i)$$

$$k_{opt} = \text{argmax}_{2\le k\le n}\{avg_{BWP}(k)\}$$

1.2.2 BWP 指标与最佳聚类数确定

周世兵等结合 k-means 算法以及式(5)定义的 BWP 聚类有效性指标，提出一种新的分析聚类效果，确定最佳聚类数的算法。算法归纳如下。

（1）选择聚类数的搜索范围[k_{min}，k_{max}]。

（2）从k_{min}循环至k_{max}:

① 调用 K-means 算法;

② 利用式(5) 计算单个样本的 BWP 指标值;

③ 利用式(6) 计算平均 BWP 指标值。

（3）利用式(7) 计算最佳聚类数。

（4）输出最佳聚类数、有效性指标值和聚类结果。

本报告用 Python 实现的代码如下：

```
# 定义在 Kmeans 类内的函数 cal_BWP 用于计算上述 BWP 指标
    # 计算 b(i,j), w(i,j) 以及 BWP(i,j)，并求出该类数 k 下的平均 BWP（avg_BWP)
    def cal_BWP(self):
        cluster = list(
            map(lambda j: np.array(list(range(self.n)))[np.where(self.point_cluster_assign == j)], list(range(self.k))))
        # 定义第 j 类的第 i 个样本的最小类间距离 b(j，i)：该样本到其他每个类中样本平均距离的最小值
        self.cluster = cluster
        b_mat = list(range(self.k))
        for j in range(self.k):
            other_list = list(range(j)) + list(range(j + 1, self.k))
            b_mat[j] = np.min(
                np.array(list(map(lambda x: np.mean(self.distance[cluster[j]][:, cluster[x]], axis=1), other_list))),
```

```
                axis=0)
        # 第 j 类的第 i 个样本的类内距离 w(j，i)：该样本到第 j 类中其他所有样本的平均距离
        w_mat = list(range(self.k))
        for j in range(self.k):
            w_mat[j] = np.mean(self.distance[cluster[j]][:, cluster[j]], axis=1)
        baw = np.array(b_mat) + np.array(w_mat)
        bsw = np.array(b_mat) - np.array(w_mat)
        BWP = bsw / baw
        self.avg_BWP = np.array(list(map(lambda i: np.mean(BWP[i]), list(range(self.k))))).mean()

    # 定义类外函数，循环求出最佳聚类数 k
def cal_best_k(data, k_list, minpt = 10, p = 60):
    BWP_list = []
    for k in k_list:
        km = Kmeans(data, minpt, p, k)
        km.cal_all()
        BWP_list.append(km.avg_BWP)
    print(BWP_list)
    return k_list[np.argmax(np.array(BWP_list))]
```

1.3 k-means 与 k-medoids

k-means 通过计算一类记录的均值来代表该类，但是受异常值或极端值的影响比较大，而另外一种算法 k-medoids 可以解决这个问题。

k-medoids 算法 k-means 十分类似，二者区别在于中心点的选取，在 k-means 中，我们将中心点取为当前 cluster 中所有数据点的平均值，而在 k-medoids 算法中，我们将从当前 cluster 中选取这样一个点——它到其他所有（当前 cluster 中的）点的距离之和最小——作为中心点。

由于 k-medoids 算法的流程和 k-means 相同，差别仅在于中心点计算的方式，因此我们可以在 Kmeans 类中将中心点计算方法作为参数输入，并添加一种计算中心点的方式，就可以使 Kmeans 类同时具有执行 k-medoids 算法的功能。

定义一个类外函数 cal_center 来计算聚类中心，代码如下：

```
# 定义类外函数 cal_center
# 在 K-means 中，我们将中心点取为当前 cluster 中所有数据点的平均值，
```

```
# 在K-medoids 算法中，我们将从当前cluster 中选取这样一个点——它到其他所有（当前cluster 中的）点的距离之和最小——作为中心点。
def cal_center(X, method = "mean"):
    if method == "mean":
        center = np.mean(X, axis=0)
    if method == "medoid":
        distance_tmp = mm_EuDistance(X,X)
        center = X[np.argmin(np.sum(distance_tmp, axis=0), axis=0)]
    return center
```

2. 测试

2.1 用模拟数据测试 Kmeans 类并选出最优 k 值

将上述 Kmeans 类以及类外函数写入 python 文件，在此调用，以测试 Kmeans 类的功能，探究最优 k 值的规律。

```
# 导入 Kmeans 类
import numpy as np
import os
os.chdir("D:/1s/数据挖掘/1 大作业/聚类/k_means")
from Kmeans import Kmeans

# 随机生成 100 个七维数据，k 的取值范围从 2 到\sqrt{100}=10
np.random.seed(5)
data = np.random.randn(100,3)
k_list = list(range(2,10))
best_k = cal_best_k(data, k_list)
print(best_k)
```

```
迭代次数为：
5
迭代次数为：
7
迭代次数为：
10
迭代次数为：
12
迭代次数为：
16
迭代次数为：
12
迭代次数为：
7
```

迭代次数为：

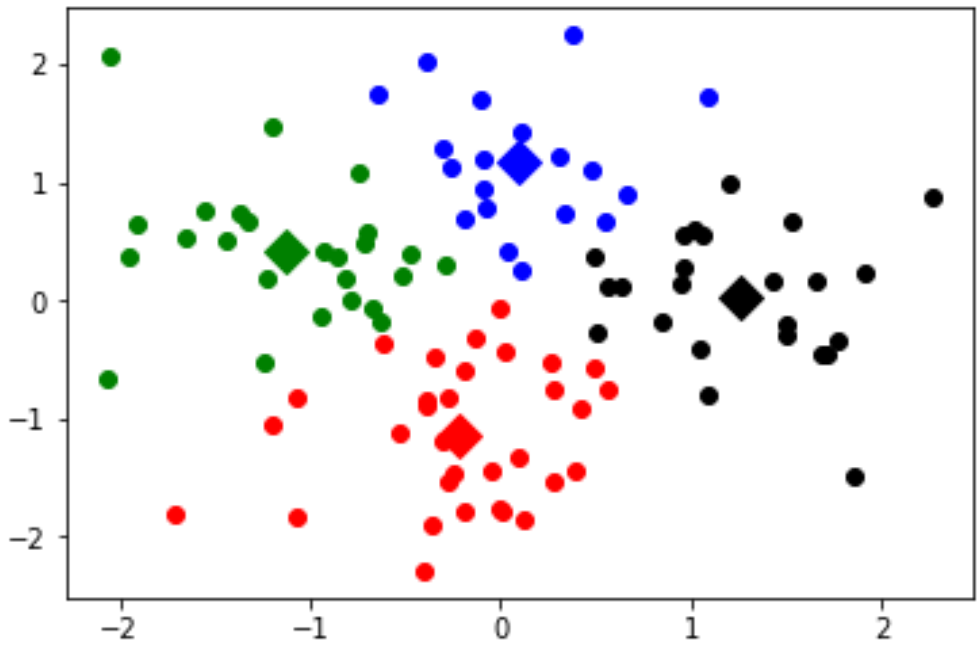
```
8
[0.14640027023289875, 0.14829840872416228, 0.18771109667504968, 0.17
724432191108216, 0.15743637945540942, 0.16124675836834815, 0.1567776
4865782978, 0.16777556346042902]
4
```

从以上结果可以看出，随着 k 的增大，BWP 指标先增大后减小，使得 BWP 取到最大值的 k 值为 4。

```
# 用二维数据画出聚类结果（取 k=4）
np.random.seed(7)
data2 = np.random.randn(100,2)
k2 = Kmeans(data2, 10, 25, 4)
k2.cal_all()
k2.plot_cluster()
```

迭代次数为：

```
7
```



2.2 用真实数据测试 Kmeans 类——对比 k-means，k-medoids

在这个部分，我选用了 UCI 数据库中的经典数据 Iris 数据 http://archive.ics.uci.edu/ml/datasets/Iris 来进行测试，将数据集按照 2:1 的比例分

为训练集和测试集，将训练集输入 Kmeans 类中，算出聚类中心，然后利用这些聚类中心对测试集数据进行类别预测。分别对 k-means 和 k-medoids 方法进行测试。

测试代码如下：

```
iris = pd.read_csv("D:/1s/数据挖掘/1 大作业/聚类/data/iris.csv")
# 生成乱序 index 来打乱数据顺序，并按照 2:1 的比例分为训练系和测试集
np.random.seed(7)
index1 = np.arange(len(iris))  # 生成下标
np.random.shuffle(index1)

iris_train = np.array(iris)[index1[:100]]
iris_test = np.array(iris)[index1[100:]]
#iris_x = iris[['sepal length in cm', 'sepal width in cm', 'petal len
gth in cm', 'petal width in cm']]
km_iris1 = Kmeans(np.array(iris_train[:, :-1]), 40, 5, 3, center_meth
od = "mean")
km_iris1.cal_all()
train_mean = km_iris1.point_cluster_assign+1
train_medoid = km_iris2.point_cluster_assign+1
print(train_mean)
km_iris2 = Kmeans(np.array(iris_train[:, :-1]), 40, 5, 3, center_meth
od = "medoid")
km_iris2.cal_all()
print(iris_test[:,-1])
test_mean = km_iris1.new_assign(iris_test[:, :-1])
test_medoid = km_iris2.new_assign(iris_test[:, :-1])
print(test_mean)
print(test_medoid)
```

迭代次数为：

```
11
[3 3 1 3 3 1 2 3 1 3 2 3 1 2 1 3 2 2 1 1 3 2 3 3 2 3 3 3 3 2 3 3 1 2 3
1 1
 1 1 2 2 3 2 2 3 1 3 3 2 1 1 1 2 1 2 3 3 3 1 1 1 3 2 3 3 1 2 1 1 2 3 1
2 1
 3 2 3 1 2 1 2 2 3 1 1 3 2 1 2 3 3 1 3 1 2 2 1 1 2 3]
```

迭代次数为：

```
6
[3. 3. 2. 1. 1. 3. 1. 1. 2. 3. 3. 2. 2. 1. 3. 1. 1. 2. 2. 3. 1. 2. 2.
3.
 3. 2. 3. 1. 2. 2. 1. 1. 1. 2. 2. 1. 3. 3. 2. 3. 1. 3. 2. 2. 1. 3. 2.
```

```
3.
 2. 1.]
[3 2 3 1 1 3 1 1 3 2 2 3 3 1 2 1 1 3 3 2 1 3 3 3 2 3 3 1 3 3 1 1 1 3 3
1 2
 3 3 2 1 2 3 3 1 3 3 2 3 1]
[2 2 3 1 1 3 1 1 2 2 2 3 3 1 2 1 1 3 3 2 1 3 3 2 2 3 2 1 2 3 1 1 1 3 3
1 2
 2 3 2 1 2 3 2 1 2 3 2 3 1]
```

```
train_mean[train_mean==3] = 4
train_mean[train_mean==2] = 3
train_mean[train_mean==4] = 2

train_medoid[train_medoid==3] = 4
train_medoid[train_medoid==2] = 3
train_medoid[train_medoid==4] = 2

accuracy_train_mean = len(np.array(range(len(iris_train)))[np.where
(train_mean == np.array(iris_train[:, -1]))]) / len(iris_train)
accuracy_train_medoid = len(np.array(range(len(iris_train)))[np.wher
e(train_medoid == np.array(iris_train[:, -1]))]) / len(iris_train)
print("iris 数据训练集—k-means 聚类准确率为：")
print(accuracy_train_mean)
print("iris 数据训练集—k-medoids 聚类准确率为：")
print(accuracy_train_medoid)
```

```
iris 数据训练集—k-means 聚类准确率为：
0.89
iris 数据训练集—k-medoids 聚类准确率为：
0.88
```

```
test_mean[test_mean==3] = 4
test_mean[test_mean==2] = 3
test_mean[test_mean==4] = 2

test_medoid[test_medoid==3] = 4
test_medoid[test_medoid==2] = 3
test_medoid[test_medoid==4] = 2

accuracy_test_mean = len(np.array(range(len(iris_test)))[np.where(te
st_mean == np.array(iris_test[:, -1]))]) / len(iris_test)
accuracy_test_medoid = len(np.array(range(len(iris_test)))[np.where
(test_medoid == np.array(iris_test[:, -1]))]) / len(iris_test)
print("iris 数据测试集—k-means 聚类准确率为：")
print(accuracy_test_mean)
```

```
print("iris 数据测试集—k-medoids 聚类准确率为：")
print(accuracy_test_medoid)
```

```
iris 数据测试集—k-means 聚类准确率为：
0.88
iris 数据测试集—k-medoids 聚类准确率为：
0.92
```

可以看出，在训练集对样本进行聚类的结果中，k-means 方法计算聚类中心的预测准确率要略高于 k-medoids 方法，而在利用训练集得出每个类别的中心后再对测试集进行类别预测的结果中，k-medoids 方法的准确率则要高于 k-means，这也符合这两个计算类别中心的方法的特点，k-medoids 本身就是对于 k-means 算法对噪音敏感缺陷的一种改进，因此能更好地避免过拟合，即在测试集上表现更好，而在训练集上则是 k-means 拟合效果更好。

2.3 自定义 Kmeans 类与 sklearn 库中 Kmeans 方法准确度比较

本部分调用 sklearn 中关于聚类的库 sklearn.cluster.KMeans 来对上述 Iris 数据集进行测试，并与本报告中的自定义 Kmeans 类的预测准确率进行对比。

代码如下：

```
from sklearn.cluster import KMeans
np.random.seed(1)
clf = KMeans(n_clusters=3, max_iter = 100)
s = clf.fit(iris_train[:, :-1])
train_predict = s.predict(iris_train[:, :-1])
test_predict = s.predict(iris_test[:, :-1])
accuracy_train_sklearn = len(np.array(range(len(iris_train)))[np.whe
re(train_predict == np.array(iris_train[:, -1]))]) / len(iris_train)
accuracy_test_sklearn = len(np.array(range(len(iris_test)))[np.where
(test_predict == np.array(iris_test[:, -1]))]) / len(iris_test)
print("sklearn 在测试集上的准确率为：")
print(accuracy_train_sklearn)
print("sklearn 在训练集上的准确率为：")
print(accuracy_test_sklearn)
```

```
sklearn 在测试集上的准确率为：
0.64
```

```
sklearn 在训练集上的准确率为：
0.68

accuracy = pd.DataFrame({'self_Kmeans':[accuracy_train_mean,accuracy
_test_mean],
                         'self_Kmedoids':[accuracy_train_medoid, accura
cy_test_medoid],
                         'sklearn_Kmeans':[accuracy_train_sklearn, accu
racy_test_sklearn]}).rename(index={0: 'train', 1: 'test'})
accuracy
```

	self_Kmeans	self_Kmedoids	sklearn_Kmeans
train	0.89	0.88	0.64
test	0.88	0.92	0.68

从以上表格中可以看出，在 Iris 数据集上，自定义的 k-means 和 k-medoids 算法在训练集和测试集预测准确度都是不错的，比 sklearn 库中提供的 k-means 聚类方法的预测准确度要高，这说明基于密度和距离的优化初始聚类中心的方法是有效的。并且由每次调用 Kmeans 类时输出的迭代次数可以看出，在样本数量在 100-200 之间（维度为 3）时，迭代次数最大为 16，目标函数值较快就达到了收敛。

3. 总结

由于 k-means 算法易于描述，具有时间效率高且适于处理大规模数据等优点，因此被广泛地应用于然语言处理等多个领域。但 k-means 算法也有很多缺陷，如需要预先确定 k 值、会受到初始聚类中心影响、难以处理分类属性数据以及容易收敛于局部最优解等。目前已经有许多研究对这些问题提出了不同的解决办法。对于选择最优 k 值，有基于聚类有效性函数、遗传算法等解决方法；对于如何确定初始中心，有基于密度和优化算法等解决方法。本实验选取了其中的两篇论文来进行算法实现，并在模拟数据和真实数据上都得到了不错的聚类结果。

本实验的亮点在于：

（1）多处运用矩阵运算，例如在求初始数据对象两两之前距离以及在 k-

medoid 方法计算聚类中心时，只进行一次矩阵运算（详见 Kmeans 类中的方法），大大减少了重复计算次数以及 for 循环的使用数量，提高了运算效率；

（2）改进 k-means 算法，选取被引次数较多的论文进行具体实现，在模拟数据和真实数据上都得到了不错的结果，相比于调用 sklearn 中 Kmeans 库的聚类结果准确度更高。

本实验的不足之处在于：

（1）仅仅选取了一个样本数量较少的数据集进行测试，对于大样本以及特征更多的数据集，自定义 Kmeans 类的聚类效果还有待检验；

（2）自定义 Kmeans 类中有一些参数，如 minpts（用以 xi 为中心，包含常数 minpts 个数据对象的半径来衡量 xi 的密度）和 p（初始中心的备选点的个数）没有进行调优；

（3）自定义 Kmeans 类虽然对 k-means 算法进行了改进并提供了两种计算聚类中心的方法，但是整体上提供的方法以及参数的设定上还比较单一，可以进一步对其进行丰富。

参考文献

[1] Sun JG, Liu J, Zhao LY. Clustering algorithms research [J]. Journal of Software, 2008, 19(1):48−61.

[2] 汪中, 刘贵全, 陈恩红. 一种优化初始中心点的K-means算法[J]. 模式识别与人工智能, 2009, 22(2): 299-304.

[3] 吴夙慧, 成颖, 郑彦宁, 潘云涛. K－ means 算法研究综述 [J]. 现代图书情报技术, 2011, 5: 28-35.

[4] 袁方, 周志勇, 宋鑫. 初始聚类中心优化的 k-means 算法 [J]. 计算机工程, 2007(33): 65-66.

[5] 周世兵, 徐振源, 唐旭清. K-means 算法最佳聚类数确定方法 [J]. 计算机应用, 2010, 30(8): 1995-1998.