

## 第 5 章 $k$ 均值聚类

---

### 一种典型的无监督学习

---

$k$ 均值聚类 ( $k$ -means clustering) 是一种将数据样本划归到指定数量的“簇” (见 1.1.2) 中的简单聚类方法。 $k$ -means 的  $k$  指的是将数据集划分为“簇”的个数。 $k$ -means 聚类是聚类算法中一种比较简单的基础算法, 是公认的十大数据挖掘算法之一。

#### 5.1 聚类分析概念

聚类分析是一种典型的无监督学习, 用于对数据样本进行划分, 将它们按照一定的规则划分成若干个“簇”。相似的样本聚类在同一个“簇”中, 不相似的样本则在不同的“簇”, 以此揭示样本的内在性质及内在规律。聚类算法在银行、零售、保险、医学、军事等领域都有着广泛的应用。

$k$ -means 聚类是基于距离度量 (见 2.1.2 节) 的聚类算法, 其基本思想是: 通过计算样本点与“簇”中心的距离, 将距离“簇”中心较近的样本点划分到该“簇”。

#### 5.2 $k$ -means 聚类算法的原理

如图 5.1 所示, 为一个样本集示例。直观上可看出, 该样本集可分为 3 个“簇”。其中, “簇”大小指“簇”中所含样本的数量; “簇”中心指一个“簇”中所有样本点的特征空间坐标均值; “簇”密度指簇中样本点的紧密程度。

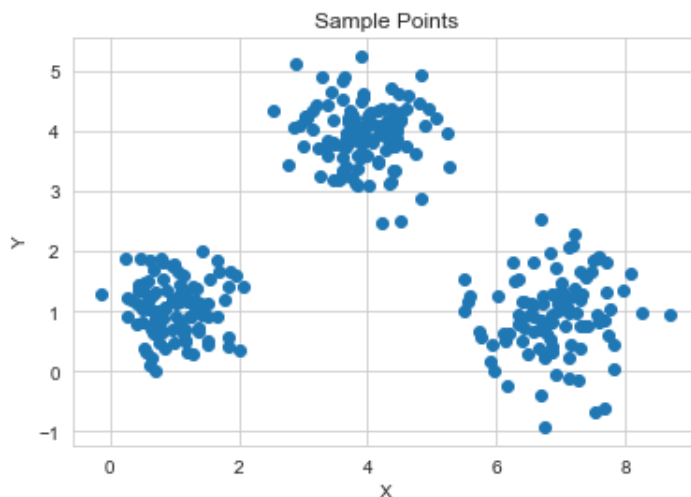
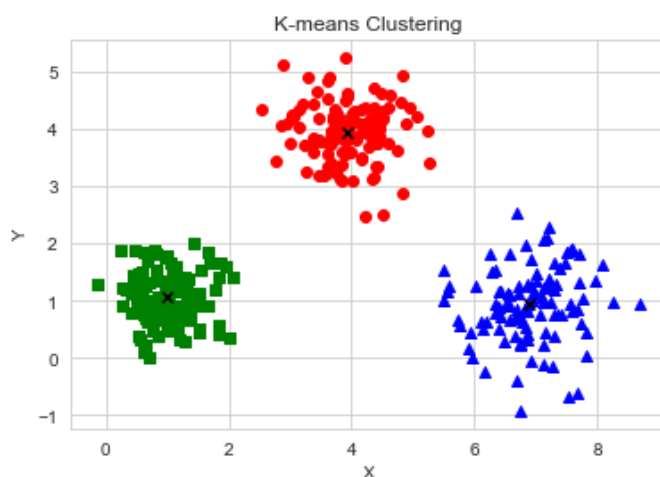


图 5.1 样本集示例

$k$ -means 聚类算法将样本划分为互斥的“簇”。通过确定  $k$  个“簇”中心的位置, 将每个样本点划归到距离最近的中心点所属的“簇”。那么, 如何确定这些中心点呢? 实际上, 一般使用的是一种基于迭代的优化方法。即从随机中心位置开始, 将样本划分给最近的中心, 然后使用这些“簇”成员的平均位置来更新中心位置。接着用这些新的中心点对样本重新进行“簇”划分。重复以上过程, 直到“簇”中心收敛到固定位置。

如图 5.2 所示, 为利用  $k$ -means 聚类算法对图 5.1 样本集的  $k=3$  聚类结果。

图 5.2  $k$ -means 的  $k=3$  聚类结果

具体来讲,  $k$ -means 聚类算法的步骤如下:

- (1) 用户指定 $k$ 的值。
- (2) 随机选取 $k$ 个样本作为  $k$  个“簇”的中心。
- (3) 对每一个待划分样本点, 计算它到各个中心的距离 (比如欧式距离), 并将其归入距离最小的“簇”中。
- (4) 根据划分情况重新计算各个“簇”的新中心。

重复执行步骤 (3) 和 (4), 直到所有样本点的划分情况保持不变, 此时说明  $k$  均值聚类已收敛到了最优解。

因为  $k$ -means 聚类算法的初始中心位置是随机选取的, 这种迭代有时会导致收敛到非最优中心位置。因此, 可以通过尝试多个起始点位置, 得到多个聚类结果。那么, 问题来了, 这些结果中, 哪一个结果是最好的呢? 直观上来讲, 一个好的聚类结果应该让样本点尽可能聚集到中心点附近。因此, 可以对每个样本点到“簇”中心的距离进行求和, 作为聚类质量的度量。这样, 从多个初始中心位置运行该算法, 最终会返回具有最低整体距离的聚类结果。

这里给出一个常用的聚类质量度量公式, 即平方误差:

$$J = \sum_{i=1}^k \|x^{(i)} - \mu_{c(i)}\|^2 \quad (5.1)$$

其中,  $\mu_{c(i)}$  表示第  $i$  个样本  $x^{(i)}$  所属簇的中心,  $J$  表示每个样本点到其所在簇中心  $\mu_{c(i)}$  距离的平方和。 $J$  越小, 所有样本点与其所在簇的整体距离越小, 样本划分的质量越好。 $k$ -means 算法的终止条件就是  $J$  收敛到最小值。但要让  $J$  收敛到最小, 需要对所有样本点可能的“簇”划分情况进行穷举, 这是一个 NP 难问题, 因此  $k$ -means 算法常采用贪心策略进行求解。

如何求得目标函数 (式 (5.1)) 的最小值呢? 首先将平方误差公式进行变形, 以一维数据为例,  $x_j$  表示第  $j$  个样本,  $c_i$  表示第  $i$  个簇的中心, 则有:

$$J = \sum_{i=1}^k \sum_{x_j \in c_i} (x_j - c_i)^2 \quad (5.2)$$

$$\begin{aligned} \frac{\partial J}{\partial c_i} &= \frac{\partial}{\partial c_i} \sum_{i=1}^k \sum_{x_j \in c_i} (x_j - c_i)^2 \\ &= \sum_{i=1}^k \sum_{x_j \in \mu_i} \frac{\partial}{\partial c_i} (x_j - c_i)^2 \\ &= -2 \sum_{x_j \in c_i} (x_j - c_i) \end{aligned} \quad (5.3)$$

当  $-2 \sum_{x_j \in c_i} (x_j - c_i) = 0$  时， $c_i = \frac{1}{|c_i|} \sum_{x_j \in c_i} x_j$ ， $|c_i|$  表示第  $i$  个簇的样本个数，即最优化的结果就是计算“簇”内样本点的均值。

在实际应用中，如果数据集过大，导致算法收敛速度过慢，而无法得到有效结果。此时，可以为  $k$ -means 聚类算法指定最大收敛次数或指定“簇”中心变化阈值，当算法运行达到最大收敛次数或“簇”中心变化率小于某个阈值时，算法即停止运行。

### 5.3 $k$ -means 聚类算法中 $k$ 的选取方式

$k$ -means 聚类算法中， $k$  值的选取方式显然影响着聚类效果。如果事先知道所有样本点中有多少个“簇”，或者对“簇”的个数有明确要求，那么在指定  $k$  值时没有太大问题。但在实际应用中，对一些数据集，很多情况下并不知道样本的分布情况，“簇”的个数选取并不能直观看出，那么这时候  $k$  应该如何选取呢？

一种方式是尝试不同的  $k$  取值。即  $k$  取不同值时，基于聚类结果计算“簇”内的  $SSE$  值，并据此画出  $SSE$  变化曲线图，从而找到最佳的  $k$  值。“簇”内的总变化量  $SSE$  定义为各个簇内的变化量之和，即：

$$SSE = \sum_{i=1}^N \sum_{j=1}^K dist(x_i, c_j) \quad (5.4)$$

其中， $N$  表示样本点的数量， $K$  表示簇的数量， $x_i$  表示第  $i$  个样本点， $c_j$  表示第  $j$  个“簇”的聚类中心， $dist(\cdot)$  表示  $x_i$  和  $c_j$  之间的距离度量函数。

以鸢尾花数据集为例。如图 5.3 所示，为鸢尾花示意图，其中花瓣（petal）

和萼片（sepal）能很好地表征鸢尾花的外观和类型。据此，鸢尾花数据集取了 4 个特征，分别为花瓣长、花瓣宽、萼片长和萼片宽。

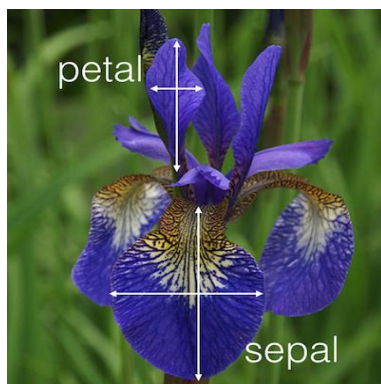


图 5.3 鸢尾花示意图

如图 5.4 所示，将鸢尾花数据集的花瓣长（横轴）和花瓣宽（纵轴）两个特征及对应的类别用散点图可视化出来，其中圆点、X 点、方点分别表示鸢尾花的三种不同类别。由该图可见，三种不同类别自然地形成了三个“簇”。因此，接下来我们尝试通过  $SSE$  变化曲线图来找到这个最佳的  $k$  值。

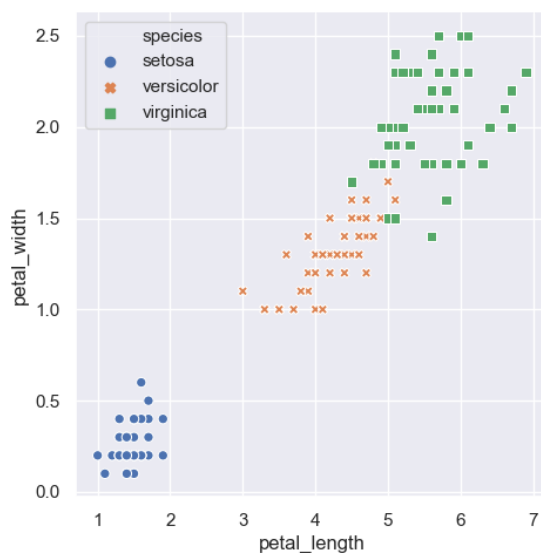


图 5.4 鸢尾花数据集散点图

依此取  $k = 1, 2, \dots, 9$ ，基于聚类结果作出“簇”内的总变化量  $SSE$  的折线图，如图 5.5 所示。由该图可见，随着  $k$  值的增加，“簇”内的总变化量总比前一次更小。

当每个“簇”内只有一个样本点时, “簇”内的总变化量为 0。但是, 从图 5.4 可知, 并不是 $k$ 值越大越好。因此, 一种常见的做法是, 采用肘部技术 (Elbow Technique) 进行 $k$ 值的选取。由图 5.5 可见, 当 $k = 3$ 时“簇”内总变化量有个较大的减小, 但之后, “簇”内总变化量就不会下降那么快了, 这就是“肘部点”, 据此可以确定最佳的 $k$ 值为 3。

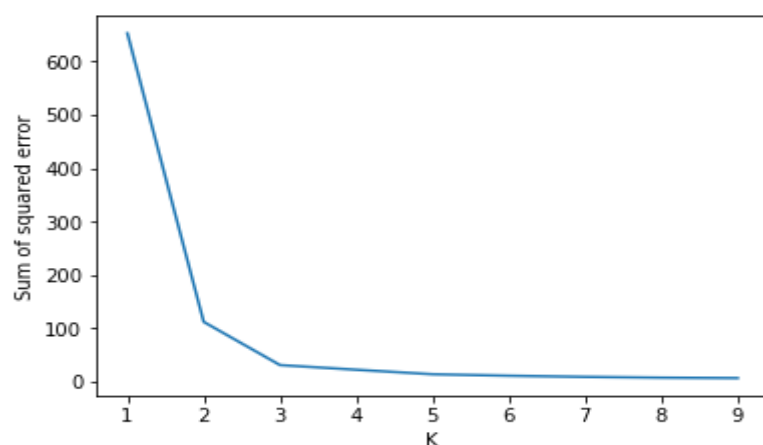


图 5.5 鸢尾花数据集 SSE 折线图

显然, 在大数据集下这样的做法非常耗费资源。因此, 各种更实用的选取最佳 $k$ 值的做法被提了出来。比如下面两种:

(1) 与层次聚类算法结合, 先通过层次聚类算法得到大致的聚类数目, 并且获得一个初始聚类结果, 然后再通过 $k$ -means 聚类算法改进这个聚类结果。

(2) 基于系统演化方法, 通过模拟伪热力学系统中的分裂和合并, 不断演化直到达到稳定平衡状态, 从而确定 $k$ 值。

#### 5.4 $k$ -means 聚类算法的优缺点

$k$ -means 聚类算法原理简单, 容易实现, 运行效率较高。算法的聚类结果容易解释, 且适用于高维数据的聚类。对于大数据集, 可指定迭代次数, 在牺牲一定准确度的情况下提升算法的运行效率。

由于  $k$ -means 聚类算法采用了贪心策略对样本进行聚类, 导致算法容易局部收敛, 在大规模的数据集上求解较慢。 $k$ -means 聚类算法对离群点和噪声点非常敏感, 少量的离群点和噪声点可能对算法求平均值产生极大影响, 从而影响最终的聚类结果。 $k$ -means 聚类算法仅在凸形“簇”结构的数据集上效果较好。此外,  $k$ -means 聚类算法在使用时, 还需注意如下问题:

- (1) 模型的输入数据为数值型数据。
- (2) 需要将原始数据作归一化或标准化处理, 因为涉及到距离计算。

### 5.5 $k$ 均值++聚类算法

$k$ -means 聚类算法的初始聚类中心的选择对算法结果影响很大, 不同的初始中心可能会导致不同的聚类结果。对此, 提出一种“ $k$ 均值++聚类算法”, 其思想是使初始聚类中心的相互距离尽可能远。步骤如下:

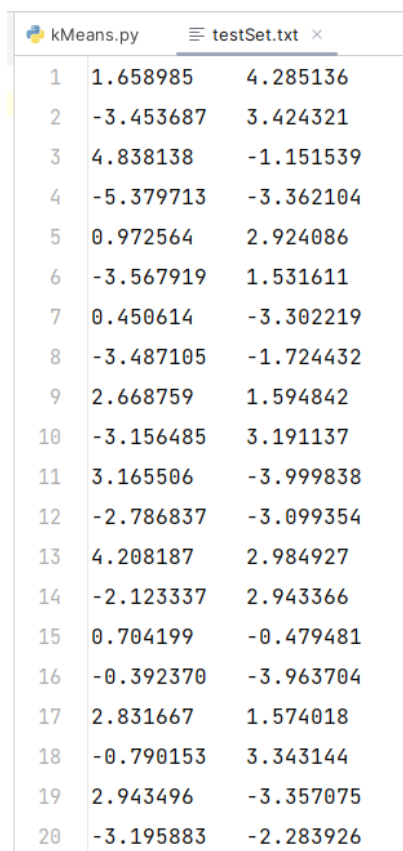
- (1) 从样本集 $\chi$ 中随机选取一个样本点 $x^{(i)}$ 作为第 1 个聚类中心;
- (2) 计算其他样本点 $x^{(j)}$ 到最近的聚类中心的距离 $d(x)$ ;
- (3) 以概率 $\frac{d(x)^2}{\sum_{x \in \chi} d(x)^2}$ 选择一个新的样本点 $x^{(l)}$ 加入聚类中心点集合中, 其中距离值 $d(x)$ 越大, 被选中的可能性越高;
- (4) 重复(2)和(3)选定 $k$ 个聚类中心;
- (5) 基于这 $k$ 个聚类中心进行 $k$ 均值运算。

### 5.6 $k$ -means 聚类的核心代码实现

本节将介绍 $k$ -means 聚类算法的 *Python* 代码实现, 并给出代码的简单运行实例。相信读者在此基础上能够在实验课中进一步完善和改进代码, 完成更具挑战性和实用性的应用任务。

如图 5.6 所示, 数据集共 80 个样本点, 每一行对应一个样本点, 每一列为

一个特征, 共两列。图中仅展示了前 20 条样本。



1	1.658985	4.285136
2	-3.453687	3.424321
3	4.838138	-1.151539
4	-5.379713	-3.362104
5	0.972564	2.924086
6	-3.567919	1.531611
7	0.450614	-3.302219
8	-3.487105	-1.724432
9	2.668759	1.594842
10	-3.156485	3.191137
11	3.165506	-3.999838
12	-2.786837	-3.099354
13	4.208187	2.984927
14	-2.123337	2.943366
15	0.704199	-0.479481
16	-0.392370	-3.963704
17	2.831667	1.574018
18	-0.790153	3.343144
19	2.943496	-3.357075
20	-3.195883	-2.283926

图 5.6 数据集

### 5.6.1 $k$ -均值聚类算法

首先看几个 $k$ -均值聚类的支持函数:

(1) `loadDataSet()`函数, 用于将文本文件导入一个列表中。文本文件每一行为 tab 键分隔的浮点数, 将每一行添加到 `dataMat` 中, 并返回 `dataMat`。`dataMat` 是一个包含许多其他列表的列表。

`def loadDataSet(fileName):` # 函数功能: 读文件并解析 TAB 分开的浮点数

`dataMat = []` # 设每行的最后一列是目标变量的值

`fr = open(fileName)`

`for line in fr.readlines():`



```
curLine = line.strip().split('\t')

fltLine = list(map(float, curLine)) #map 将所有值转换为浮点数

dataMat.append(fltLine)

return dataMat
```

(2) `distEuclid()`函数, 用于计算两个向量的欧式距离。

```
def distEuclid(vecA, vecB):

    return np.sqrt(np.sum(np.power(vecA - vecB, 2))) #等价于

np.linalg.norm(vecA-vecB)
```

(3) `randCent()`函数, 为给定数据集构建一个包含 $k$ 个随机质心的矩阵。随机质心必须在整个数据集的边界之内, 这可以通过找到数据集每一维特征的最小和最大值来完成: 生成0到1.0之间的随机数, 通过取值范围和最小值, 确保随机点在数据的边界之内。`rangeJ` 表示 $k$ 个质心向量的第 $j$ 维数据值为位于(最小值, 最大值)区间内的某一值, `centroids[:, j]`为簇矩阵的第 $j$ 列, `random.rand(k, 1)`表示产生 $(k, 1)$ 维的随机数矩阵, 且随机数分布在半开区间 $[0, 1)$ 。

```
def randCent(dataSet, k):

    n = np.shape(dataSet)[1]

    centroids = np.mat(zeros((k, n))) #创建初值为 0 的簇中心矩阵

    for j in range(n): #创建随机簇中心, 其每一维都有取值范围

        minJ = min(dataSet[:, j])

        rangeJ = float(max(dataSet[:, j]) - minJ)

        centroids[:, j] = np.mat(minJ + rangeJ * np.random.rand(k, 1))

    return centroids
```

将以上函数保存到文件 kMeans.py 里，然后在 Python 解释器里执行如下命令：

```
>>> import kMeans

>>> import numpy as np

>>> datMat=np.mat(kMeans.loadDataSet('testSet.txt'))

>>> min(datMat[:,0])

matrix([[ -5.379713]])

>>> min(datMat[:,1])

matrix([[ -4.232586]])

>>> max(datMat[:,0])

matrix([[ 4.838138]])

>>> max(datMat[:,1])

matrix([[ 5.1904]])

>>> kMeans.randCent(datMat, 2)

matrix([[ 2.20313084, -1.69216831],

        [-2.73759393,  0.9952749 ]])

>>> kMeans.distEuclid(datMat[0], datMat[1])

5.184632816681332
```

执行结果验证了以上三个函数的正确性。

接下来就是  $k$ -均值聚类算法的具体实现——函数 kMeans()，该函数接受4个输入参数，只有数据集及簇的数目是必选参数，而用来计算距离和创建初始质心的函数都是可选的，默认为上面定义的 distEuclid()函数和 randCent()函数。

kMeans() 函数一开始确定数据集中数据点的总数，然后创建一个矩阵 clusterAssment 来存储每个点的簇分配结果。该矩阵包含两列：第一列存储簇索引值，即每个样本对应的簇中心；第二列存储误差，即当前样本点到簇中心的距离，可用来评价聚类的效果。

该函数按照“计算质心-分配-重新计算”反复迭代，直到所有数据点的簇分配结果不再改变为止：定义了一个标志变量 clusterChanged，如果该值为 True，则继续迭代。接下来遍历所有数据点，找到距离每个点最近的质心：通过对每个点遍历所有质心并计算点到每个质心的距离来完成。计算距离是调用 distMeas 参数给出的距离函数，默认是 distEuclid()。如果任一点的簇分配结果发生改变，则将 clusterChanged 标志置为 True。接下来，遍历所有质心并更新它们的取值。具体步骤为：首先通过数组过滤获得给定簇的所有点，然后计算所有点的均值，选项 axis=0 表示沿矩阵的列方向（对应特征）进行均值计算。最后，程序返回所有的簇质心及数据点分配结果。

```
def kMeans(dataSet, k, distMeas=distEuclid, createCent=randCent):
```

```
#dataset: 数据集; k: 簇的个数; distMeas: 距离计算函数; createCent: 随机  
质心生成函数
```

```
    m = np.shape(dataSet)[0] #获取数据集样本点数
```

```
    clusterAssment = np.mat(np.zeros((m, 2))) #创建一个初值为 0 的矩阵来存储  
每个数据点的簇分配结果
```

```
    centroids = createCent(dataSet, k)
```

```
    clusterChanged = True
```

```
    while clusterChanged:
```

---

```
clusterChanged = False

for i in range(m): #将每个数据点分配给最近的簇

    minDist = np.inf; minIndex = -1 #初始化最小距离为正无穷；最小
    距离对应簇索引为-1； minDist 保存最小距离, minIndex 保存最小距离对应的簇
    质心

    for j in range(k): #循环 k 个簇的质心，找到距离第 i 个样本最近的
    簇

        distJI = distMeas(centroids[j, :], dataSet[i, :]) #计算数据点到簇
        质心的欧氏距离

        if distJI < minDist:

            minDist = distJI; minIndex = j #更新当前最小距离及对应
            的簇索引

    if clusterAssment[i, 0] != minIndex: clusterChanged = True #如果第
    i 个样本点的聚类结果发生变化，则将 clusterChanged 置为 true

    clusterAssment[i, :] = minIndex, minDist**2 #更新样本点 i 的聚类结
    果

    print (centroids)

    for cent in range(k): #重新计算簇中心

        ptsInClust = dataSet[np.nonzero(clusterAssment[:,0].A==cent)[0]] #
        获取簇 cent 的所有数据点

        centroids[cent, :] = np.mean(ptsInClust, axis=0) #所有数据点的均
        值即为簇中心
```

```
return centroids, clusterAssment
```

在 Python 解释器里执行如下命令：

```
>>> myCentroids, clustAssing = kMeans.kMeans(datMat, 4)

[[ 4.23481803  2.37662322]

 [-1.42590616  1.15020136]

 [ 0.25053297 -3.03194382]

 [ 2.90651356 -2.10638585]]

[[ 2.71358074  3.11839563]

 [-2.44978374  2.38092765]

 [-2.73649319 -2.99246324]

 [ 3.17437012 -2.75441347]]

[[ 2.6265299   3.10868015]

 [-2.46154315  2.78737555]

 [-3.38237045 -2.9473363 ]

 [ 2.80293085 -2.7315146 ]]
```

可见，经过3次迭代之后 $k$ -means 聚类算法即收敛。注意，由于 randCent() 函数具有随机性，因此 kMeans()函数每次执行的结果都有所不同。图 5.7 给出了某一次执行结果的可视化，包括4个簇（分别对应 4 种形状）及其中心（图中的五角星）。

图 5.7  $k$ -means 聚类的某一次结果

### 5.6.2 二分 $k$ -均值聚类算法

一种用于度量聚类效果的指标是式 (5.4) 给出的  $SSE$  (Sum of Squared Error, 误差平方和), 其对应 `kMeans()` 函数中的 `clusterAssment` 矩阵的第一列之和。 $SSE$  值越小表示数据点越接近它们的质心, 聚类效果也越好。由于对误差取了平方, 因此更加重视远离中心的点。聪明的读者可能马上会想到, 增加簇的个数可以降低  $SSE$ , 但这样做违背了聚类的目标: 保持簇数目不变的情况下提高簇的质量。

为克服  $k$ -means 聚类算法收敛于局部最小值的问题, 提出二分  $k$ -means 聚类算法。该算法首先将所有点作为一个簇, 然后将该簇一分为二。之后选择其中一个簇继续划分, 具体选择哪一个簇进行划分取决于对其划分是否可最大程度降低  $SSE$ 。上述基于  $SSE$  的划分过程不断重复, 直到得到用户指定的簇数目为止。

函数 `biKmeans()` 是二分  $k$ -means 聚类算法的具体实现, 在给定数据集、簇数目及距离计算方法的条件下, 返回聚类结果。该函数首先创建一个矩阵来存储数据集中每个点的簇分配结果及平方误差, 然后计算整个数据集的质心, 并使用

一个列表来保留所有的质心。得到上述质心后, 遍历数据集中的所有点来计算每个点到质心的误差值。

接下来进入 while 循环, 该循环不停地对簇进行划分, 直到得到指定的簇数目为止。内层 for 循环遍历所有的簇来决定最佳的簇进行划分, 为此需要比较划分前后的  $SSE$ 。开始时, 将  $SSE$  最小值 `lowestSSE` 的初值设置为无穷大, 然后遍历簇列表 `centList` 中的每一个簇。对每个簇  $i$ , 获取到其所有数据点 `ptsInCurrCluster`, 然后调用 `kMeans()` 函数进行二分 ( $k = 2$ ), 返回两个质心, 同时给出每个簇的误差值。这些误差 (`sseSplit`) 与剩余数据集的误差 (`sseNotSplit`) 之和作为本次划分的总误差, 如果其小于当前的  $SSE$  最小值 `lowestSSE`, 则本次划分被保存。

一旦内层 for 循环决定了要划分的簇, 接下来就是执行划分操作: 将要划分的簇所有点的簇分配结果进行修改。因为调用 `kMeans()` 函数进行二分 ( $k = 2$ ), 会得到两个编号分别为 0 和 1 的簇, 所以只需要将这些簇编号修改为划分簇及新加簇的编号即可, 该过程通过两个数组过滤器实现。最后, 新的簇分配结果被保存, 新的质心被添加到 `centList` 中。

当 while 循环结束时, 函数返回质心列表与簇分配结果。

```
def biKmeans(dataSet, k, distMeas=distEuclid):  
  
    m = np.shape(dataSet)[0]  
  
    clusterAssment = np.mat(np.zeros((m, 2))) # 创建一个初值为 0 的矩阵来存储  
    数据集中每个点的簇分配结果及平方误差  
  
    centroid0 = np.mean(dataSet, axis=0).tolist()[0]  
    centList = [centroid0] # 用列表存储簇中心, 初始状态只有一个簇  
  
    for j in range(m): # 计算初始误差
```

---

```

clusterAssment[j, 1] = distMeas(np.mat(centroid0), dataSet[j,:])**2

while (len(centList) < k):

    lowestSSE = np.inf

    for i in range(len(centList)):

        ptsInCurrCluster = dataSet[np.nonzero(clusterAssment[:,
0].A==i)[0], :] #获取簇 i 的所有数据点

        centroidMat, splitClustAss = kMeans(ptsInCurrCluster, 2, distMeas)

        sseSplit = np.sum(splitClustAss[:, 1])

        sseNotSplit = np.sum(clusterAssment[np.nonzero(clusterAssment[:,0].A!=i)[0], 1])

        print("sseSplit, and notSplit: ", sseSplit, sseNotSplit)

        if (sseSplit + sseNotSplit) < lowestSSE: #本次划分的误差更小?

            bestCentToSplit = i

            bestNewCents = centroidMat

            bestClustAss = splitClustAss.copy()

            lowestSSE = sseSplit + sseNotSplit

        bestClustAss[np.nonzero(bestClustAss[:, 0].A == 1)[0], 0] = len(centList)

#将簇编号修改为划分簇及新加簇的编号

        bestClustAss[np.nonzero(bestClustAss[:, 0].A == 0)[0], 0] =
bestCentToSplit

        print('the bestCentToSplit is: ',bestCentToSplit)

        print('the len of bestClustAss is: ', len(bestClustAss))

```



```
centList[bestCentToSplit] = bestNewCents[0, :].tolist()[0] #更新为两个新的最佳簇中心
```

```
centList.append(bestNewCents[1, :].tolist()[0])
```

```
clusterAssment[np.nonzero(clusterAssment[:, 0].A == bestCentToSplit)[0, :]= bestClustAss #更新为新的簇分配结果
```

```
return np.mat(centList), clusterAssment
```

在 Python 解释器里执行如下命令：

```
>>> datMat3=np.mat(kMeans.loadDataSet('testSet2.txt'))
```

```
>>> centList,myNewAssments=kMeans.biKmeans(datMat3, 3)
```

```
[[ -4.42959232 -0.32507779]
```

```
 [ 3.58077101  2.13138021]]
```

```
[[ -1.73028592  0.20133246]
```

```
 [ 2.76275171  3.12704005]]
```

```
[[ -1.70351595  0.27408125]
```

```
 [ 2.93386365  3.12782785]]
```

```
sseSplit, and notSplit: 541.2976292649145 0.0
```

```
the bestCentToSplit is: 0
```

```
the len of bestClustAss is: 60
```

```
[[ -3.74083326 -0.30787434]
```

```
 [-1.80975894  4.17627223]]
```

```
[[ -0.74459109 -2.39373345]
```

```
 [-2.87553522  3.53474367]]
```

```
[[ -0.45965615 -2.7782156 ]  
  
 [ -2.94737575  3.3263781 ]]  
  
sseSplit, and notSplit:  67.2202000797829 39.52929868209309  
  
[[ 4.26595061  4.3663869 ]  
  
 [ 2.90421006  1.83129415]]  
  
[[ 3.6690305  4.03686067]  
  
 [ 2.61879214  2.73824236]]  
  
sseSplit, and notSplit:  27.813776175385765 501.7683305828214  
  
the bestCentToSplit is:  0  
  
the len of bestClustAss is:  40  
  
>>> centList  
  
matrix([[ -0.45965615, -2.7782156 ],  
        [ 2.93386365,  3.12782785],  
        [ -2.94737575,  3.3263781 ]])
```

可见, 运行结果恰如所预期的, 通过两次划分, 得到三个簇。多次运行, 聚类的结果相同。图 5.8 给出了 3 个簇及其中心的可视化结果。

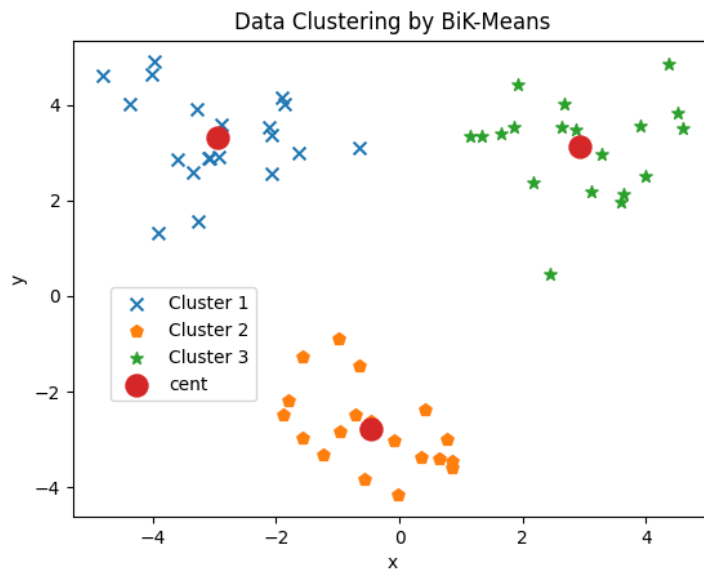


图 5.8 二分 $k$ -means 聚类结果

## 习题 5

5.1  $k$ 均值聚类算法的目标是：（ ）

- (a) 最小化簇内的平方误差和
- (b) 最大化簇间的平方误差和
- (c) 最小化簇内的距离和
- (d) 最大化簇间的距离和

5.2  $k$ 均值聚类算法的收敛条件是：（ ）

- (a) 簇内的平方误差和不再减小
- (b) 簇间的平方误差和不再减小
- (c) 簇内的距离和不再减小
- (d) 簇间的距离和不再减小

5.3 在 $k$ 均值聚类算法中，如何确定最优的簇数 $k$ ？（ ）

- (a) 根据领域知识和经验进行选择
- (b) 使用肘部法则（Elbow method）分析簇内平方误差和
- (c) 使用轮廓系数（Silhouette coefficient）评估聚类效果
- (d) 所有选项都可能

5.4  $k$ 均值聚类算法属于以下哪种类型的机器学习算法？（ ）

- (a) 监督学习
- (b) 无监督学习
- (c) 半监督学习
- (d) 强化学习

5.5 请详细说明 5.6.1 节中命令“`kMeans.randCent(datMat, 2)`”的返回值确实各特征的取值范围之内。

5.6 关于初始簇中心的选取，5.6 节的代码实现与 5.2 节的原理介绍有何不同？你如何看待这种实现上的不同？