《机器学习实战》源码解析

机器学习源码解析VIII

KMeans 算法

jackycaf

2014/5/3 Saturday

目录

第一章 数据导入和预处理数据

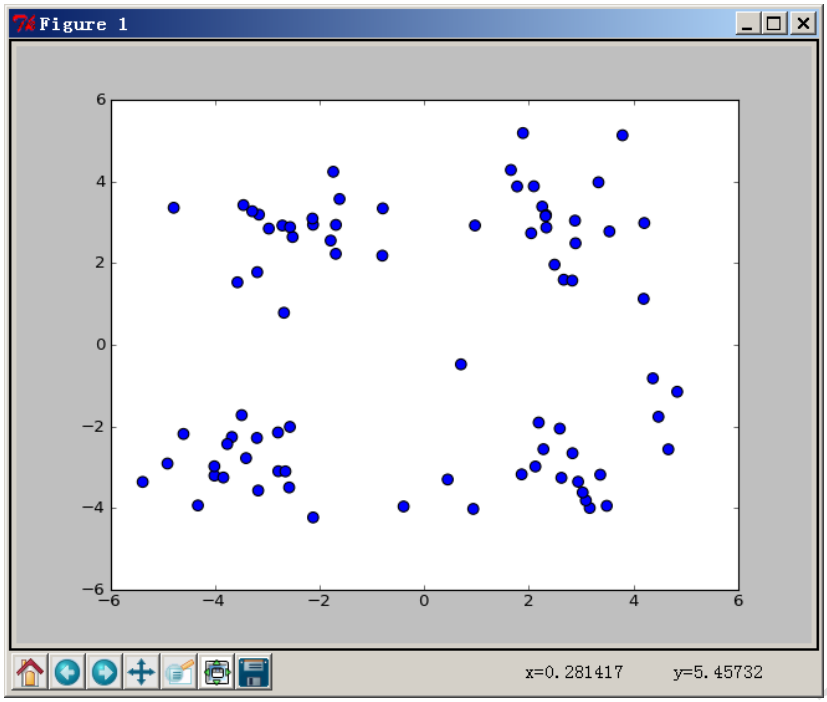
在讲解算法之前，需要简单介绍一下数据导入和预处理函数。

数据导入函数

代码区：01DateSet.py

```
# -*- coding: utf-8 -*-
# Filename : 01DateSet.py

from numpy import *
import numpy as np
import operator
import kMeans2
import matplotlib.pyplot as plt

# 导入数据集
# 将格式转换为矩阵
dataMat = []
fr = open("testSet.txt")
for line in fr.readlines():
        curLine = line.strip().split('\t')
        fltLine = map(float,curLine) # map all elements to float()
        dataMat.append(fltLine)

dataMat = mat(dataMat)     # 转换为 numpy 矩阵
for cent in dataMat:
        plt.scatter(cent[0,0],cent[0,1],s=60,c='blue',marker='o')
plt.show()
```

输出：

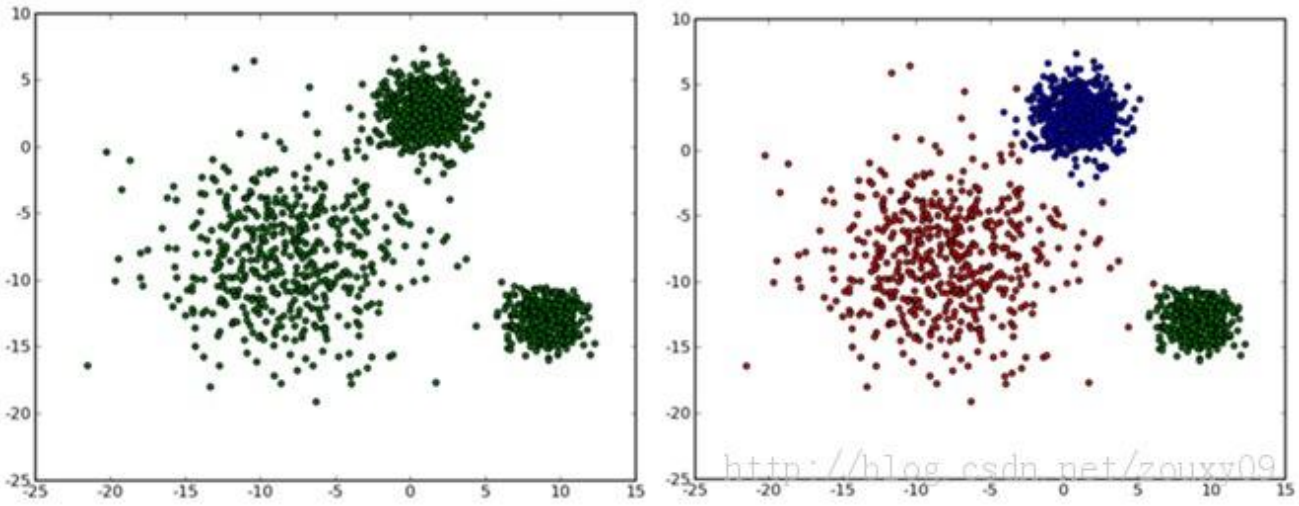


第二章 KMeans 算法

KMeans 算法理论

通常，人们根据样本间的某种距离或者相似性来定义聚类，即把相似的（或距离近的）样本聚为同一类，而把不相似的（或距离远的）样本归在其他类。

我们以一个二维的例子来说明下聚类的目的。如下图左所示，假设我们的 n 个样本点分布在图中所示的二维空间。从数据点的大致形状可以看出它们大致聚为三个 cluster，其中两个紧凑一些，剩下那个松散一些。我们的目的是为这些数据分组，以便能区分出属于不同的簇的数据，如果按照分组给它们标上不同的颜色，就是像下图右边的图那样：



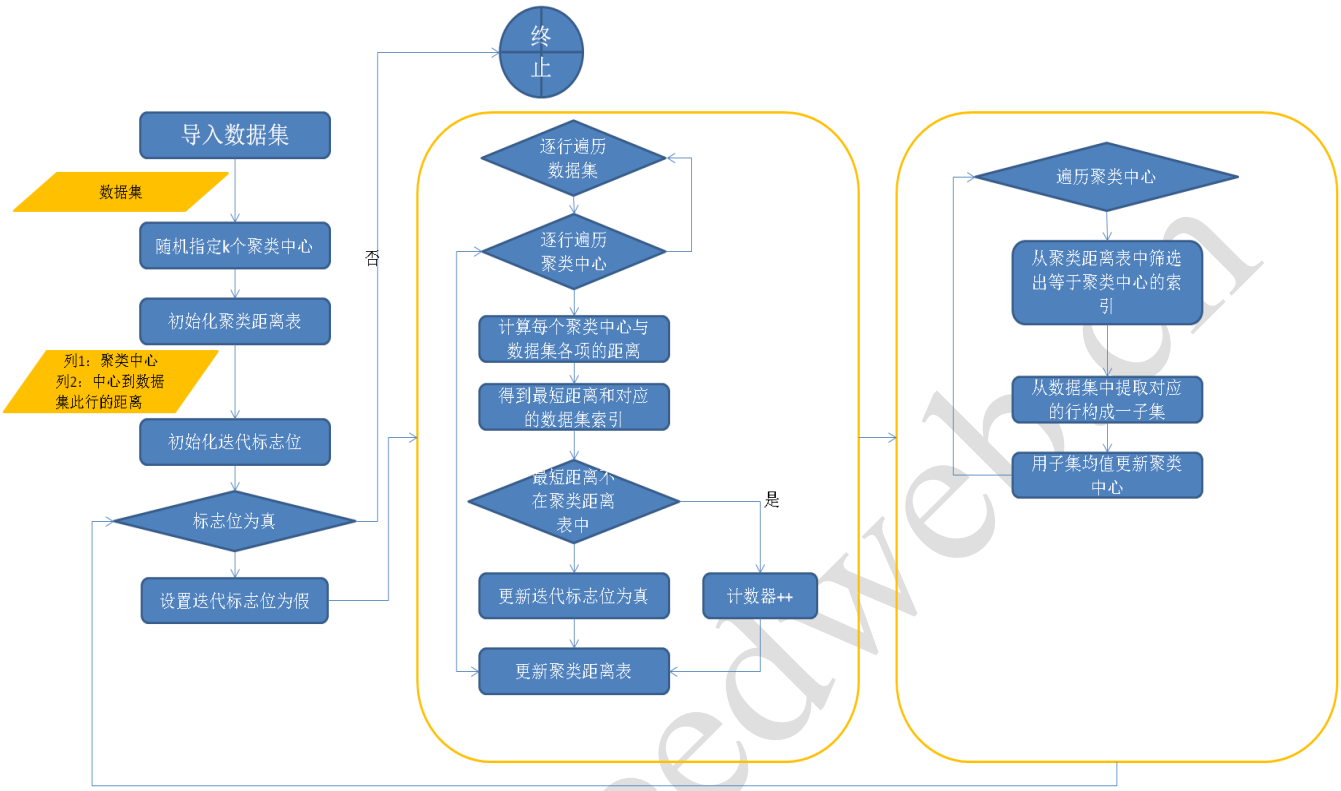
如果人可以看到像上图那样的数据分布，就可以轻松进行聚类。但我们怎么教会计算机按照我们的思维去做同样的事情呢？这里就介绍个集简单和经典于一身的 k-means 算法。

K-means 算法接受参数 k ；然后将事先输入的 n 个数据对象划分为 k 个聚类以便使得所获得的聚类满足：同一聚类中的对象相似度较高；而不同聚类中的对象相似度较小。聚类相似度是利用各聚类中对象的均值所获得一个“中心对象”（引力中心）来进行计算的。

K-means 算法是最为经典的基于划分的聚类方法，也是十大经典数据挖掘算法之一。K-means 算法的基本思想是：以空间中 k 个点为中心进行聚类，对最靠近他们的对象归类。通过迭代的方法，逐次更新各聚类中心的值，直至得到最好的聚类结果。

该算法的最大优势在于简洁和快速。算法的关键在于初始中心的选择和距离公式。

KMeans 算法流程



KMeans 主程序

```
# -*- coding: utf-8 -*-
# Filename : 02kMeans1.py

from numpy import *
import numpy as np
import operator
import kMeans2
import matplotlib.pyplot as plt

# 从文件构建的数据集
dataSet = kMeans2.loadDataSet("testSet.txt")
dataSet = mat(dataSet) # 转换为矩阵形式

# 绘制散点图图形
for mydata in dataSet:
```

```
    plt.scatter(mydata[0,0],mydata[0,1],c='blue',marker='o')

k = 4     # 外部指定 通过观察数据集有 4 个聚类中心
m = shape(dataSet)[0]     # 返回矩阵的行数

# 本算法核心数据结构:行数与数据集相同
# 列 1：数据集对应的聚类中心,列 2:数据集行向量到聚类中心的距离
clusterAssment = mat(zeros((m,2)))

# 随机生成一个数据集的聚类中心:本例为 2*4 的矩阵
# 确保该聚类中心位于 min(dataMat[:,j]),max(dataMat[:,j])之间
centroids = kMeans2.randCent(dataSet, k)

clusterChanged = True # 初始化标志位,迭代开始
counter = []; #计数器

# 循环迭代直至终止条件为 False
# 算法停止的条件：dataSet 的所有向量都能找到某个聚类中心,到此中心的距离均小于其他 k-1 个中心的距离
while clusterChanged:
    x1 = 0; x2 = 0;temp = [] #计数参数
    clusterChanged = False # 恢复标志位为 False

    #--- 1. 构建 clusterAssement： 遍历 DataSet 数据集,计算 DataSet 每行与 centroids 矩阵的最小欧式距离 ----#
    # 以及对应的 centroids 行索引，将此结果赋值 clusterAssement=[minIndex,minDist]
    for i in range(m):
        minDist = inf # 初始化最小欧式距离为无穷大
        minIndex = -1 # 初始化最小 centroids 索引值-1

        # 遍历 k 个聚类中心
        # 计算每个 dataSet 行向量与 k 个 centroids 行向量之间的欧式距离
        # 选出距离最短一行的 centroids 索引赋给 minIndex,最短距离赋给 minDist
        for j in range(k):
            # 计算 dataSet 第 i 行, centroids 第 j(0~k-1)行的欧式距离
            # distJI = kMeans2.distEclud(centroids[j,:],dataSet[i,:])
            distJI = sum(power(centroids[j,:] - dataSet[i,:], 2))

            # 如果计算的 distJI 欧式距离小于最小距离
            if distJI < minDist:
                minDist = distJI    # distJI->minDist
                minIndex = j          # j->minIndex j 是 centroids 的行下标
```

```
            # 如果 clusterAssment 当前的索引值不等于最小索引值
            if clusterAssment[i,0] != minIndex:
                clusterChanged = True #  重置标志位为 True，继续迭代
                x1 +=1              # x1 记录了不等的数量
            else: x2 +=1            # x2 记录了相等的数量

            # 将 minIndex 和 minDist**2 赋予 clusterAssement 第 i 行
            # 含义是数据集 i 行对应的聚类中心为 minIndex,最短距离为 minDist
            clusterAssment[i,:] = minIndex,minDist         # **2

        # 累计计数
        temp.append(x1); temp.append(x2); counter.append(temp) #计数器累计

        # ---- 2.更新 centroids 聚类中心: 循环变量为 cent(0~k-1)----#
        # 用聚类中心 cent 切分为 clusterAssment，返回 dataSet 的行索引
        # 并以此从 dataSet 中提取对应的行向量构成新的 ptsInClust
        # 计算分隔后 ptsInClust 各列的均值，以此更新聚类中心 centroids 的各项值
        for cent in range(k):
            # 从 clusterAssment 的第一列中筛选出等于 cent 值的行下标
            #   clusterAssment[:,0].A==cent: 判断 clusterAssment 第 1 列每个元素是否与 cent 相等,如果相等返回为 1,否则为 0
            #   nonzero(clusterAssment[:,0].A==cent):返回 clusterAssment[:,0]非零值的索引，
            #   二维的情况为两个 List:List1(行下标数组);List2(列下标数组)
            #   print clusterAssment[:,0].A
            #   print cent
            dInx = nonzero(clusterAssment[:,0].A==cent)[0]
            # 从 dataSet 中提取行下标==dInx 构成一个新数据集
            ptsInClust = dataSet[dInx]
            # 计算 ptsInClust 各列的均值: mean(ptsInClust, axis=0):axis=0 从第一列开始
            # =[ptsInClust 第 1 列所有值之和/ptsInClust 行数, ptsInClust 第 2 列所有值之和/ptsInClust 行数...]
            centroids[cent,:] = mean(ptsInClust, axis=0)

# 绘制聚类中心
for cent in centroids:
    plt.scatter(cent[0,0],cent[0,1],s=60,c='red',marker='D')
plt.show()

# 返回计算完成的聚类中心
print centroids
# 输出生成的 clusterAssment：对应的聚类中心(列 1),到聚类中心的距离(列 2),行与 dataSet 一一对应
print clusterAssment
# 计数器：统计 clusterAssment 每次循环后 minIndex 的变化
print counter
```

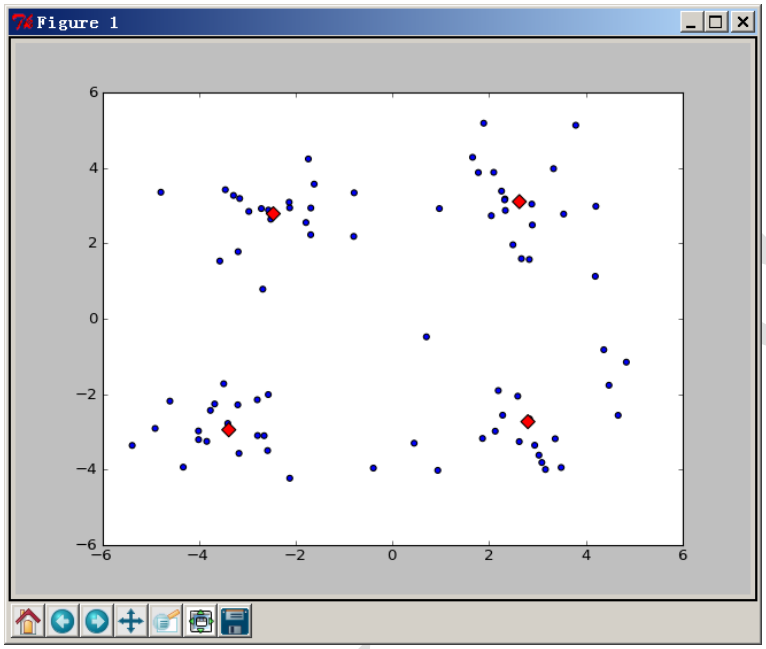
输出:

```
centroids: # 计算后得到 4 个聚类中心向量
[[-2.46154315  2.78737555]
 [ 2.6265299   3.10868015]
 [ 2.65077367 -2.79019029]
 [-3.53973889 -2.89384326]]
clusterAssment:  # 数据集对应的最近聚类中心，及其到最近聚类中心的距离
[[  1.          2.3201915 ]
 [  0.          1.39004893]
 [  2.          7.46974076]
 [  3.          3.60477283]
 [  1.          2.7696782 ]
 [  0.          2.80101213]
 [  2.          5.10287596]
 [  3.          1.37029303]
 [  1.          2.29348924]
 [  0.          0.64596748]
 [  2.          1.72819697]
 [  3.          0.60909593]
 [  1.          2.51695402]
 [  0.          0.13871642]
 [  2.          9.12853034]
 [  2.         10.63785781]
 [  1.          2.39726914]
 [  0.          3.1024236 ]
 [  2.          0.40704464]
 [  3.          0.49023594]
 [  1.          0.13870613]
 [  0.          0.510241  ]
 [  2.          0.9939764 ]
 [  3.          0.03195031]
 [  1.          1.31601105]
 [  0.          0.90820377]
 [  2.          0.54477501]
 [  3.          0.31668166]
 [  1.          0.21378662]
 [  0.          4.05632356]
 [  2.          4.44962474]
 [  3.          0.41852436]
 [  1.          0.47614274]
```

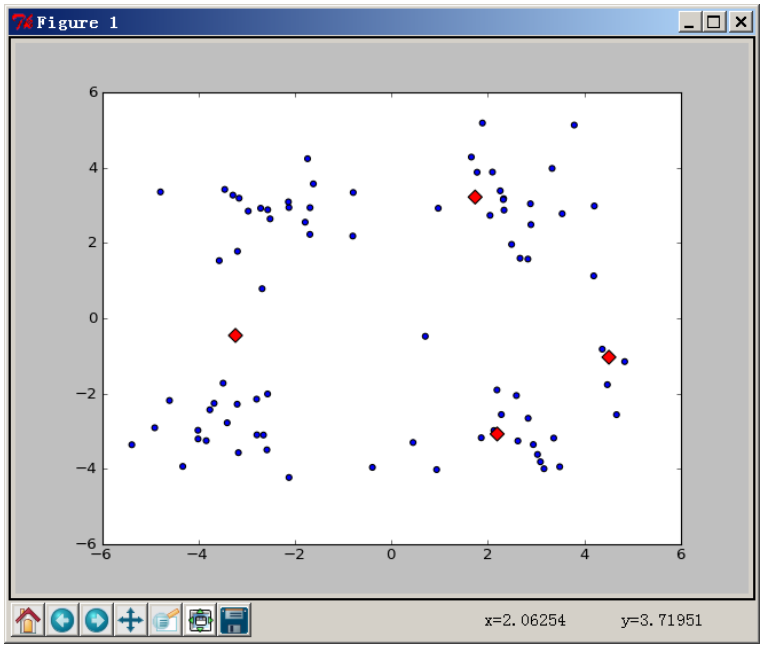
```
[ 0.        1.5441411 ]
[ 2.        6.83764117]
[ 3.        1.28690535]
[ 1.        4.87745774]
[ 0.        3.12703929]
[ 2.        0.05182929]
[ 3.        0.21846598]
[ 1.        0.8849557 ]
[ 0.        0.0798871 ]
[ 2.        0.66874131]
[ 3.        3.80369324]
[ 1.        0.09325235]
[ 0.        0.91370546]
[ 2.        1.24487442]
[ 3.        0.26256416]
[ 1.        0.94698784]
[ 0.        2.63836399]
[ 2.        0.31170066]
[ 3.        1.70528559]
[ 1.        5.46768776]
[ 0.        5.73153563]
[ 2.        0.22210601]
[ 3.        0.22758842]
[ 1.        1.32864695]
[ 0.        0.02380325]
[ 2.        0.76751052]
[ 3.        0.59634253]
[ 1.        0.45550286]
[ 0.        0.01962128]
[ 2.        2.04544706]
[ 3.        1.72614177]
[ 1.        1.2636401 ]
[ 0.        1.33108375]
[ 2.        0.19026129]
[ 3.        0.83327924]
[ 1.        0.09525163]
[ 0.        0.62512976]
[ 2.        0.83358364]
[ 3.        1.62463639]
[ 1.        6.39227291]
[ 0.        0.20120037]
[ 2.        4.12455116]
[ 3.        1.11099937]
```

```
 [    1.          0.07060147]
 [    0.          0.2599013 ]
 [    2.          4.39510824]
 [    3.          1.86578044]]
counter: # 运行了 12 次才最终得到聚类中心，第一列记录了不等的数量，第二列记录相等的数量
[[57, 23], [6, 74], [4, 76], [3, 77], [3, 77], [2, 78], [2, 78], [3, 77], [9, 71], [8, 72], [6, 74], [0, 80]]
```

算法稳定的情况：



算法不稳定的情况：



随机生成聚类中心函数

代码：kmeans2.py

```
def randCent(dataSet, k):
    n = shape(dataSet)[1]
    centroids = mat(zeros((k,n)))    # 初始化聚类中心矩阵:k*n
    for j in range(n):#create random cluster centers, within bounds of each dimension
        minJ = min(dataSet[:,j])    # 返回第 j 列的最小值
        # 范围：返回第 j 列最大值与最小值的差
        rangeJ = float(max(dataSet[:,j]) - minJ)
        # random.rand(k,1):产生一个 0~1 之间的随机数向量
        # k,1 表示产生 k 行 1 列的随机数
        # 计算第 j 列的中心: j 列最小值 + 范围值*随机数(0~1)
        centroids[:,j] = mat(minJ + rangeJ * random.rand(k,1))
    return centroids
```

计算向量间欧氏距离函数

代码：kmeans2.py

```
def distEclud(vecA, vecB):
    return sqrt(sum(power(vecA - vecB, 2))) #计算两向量的欧氏距离
```

第三章 二分 KMeans 算法

Kmeans 算法的问题

本章的主要改进了 K-means 算法本身存在的一些不足。K-means 聚类问题是一个 NP 问题(难解的非指数问题)，即问题的复杂度随着比特位数的增长而指数上升，它与其他诸多的聚类和位置问题息息相关。但是该算法却有以下缺点：

1）算法的初始中心点选择与算法的运行效率密切相关，而随机选取中心点有可能导致迭代次数很大或者限于某个局部最优状态；

2）只擅长于处理球状分布的数据；

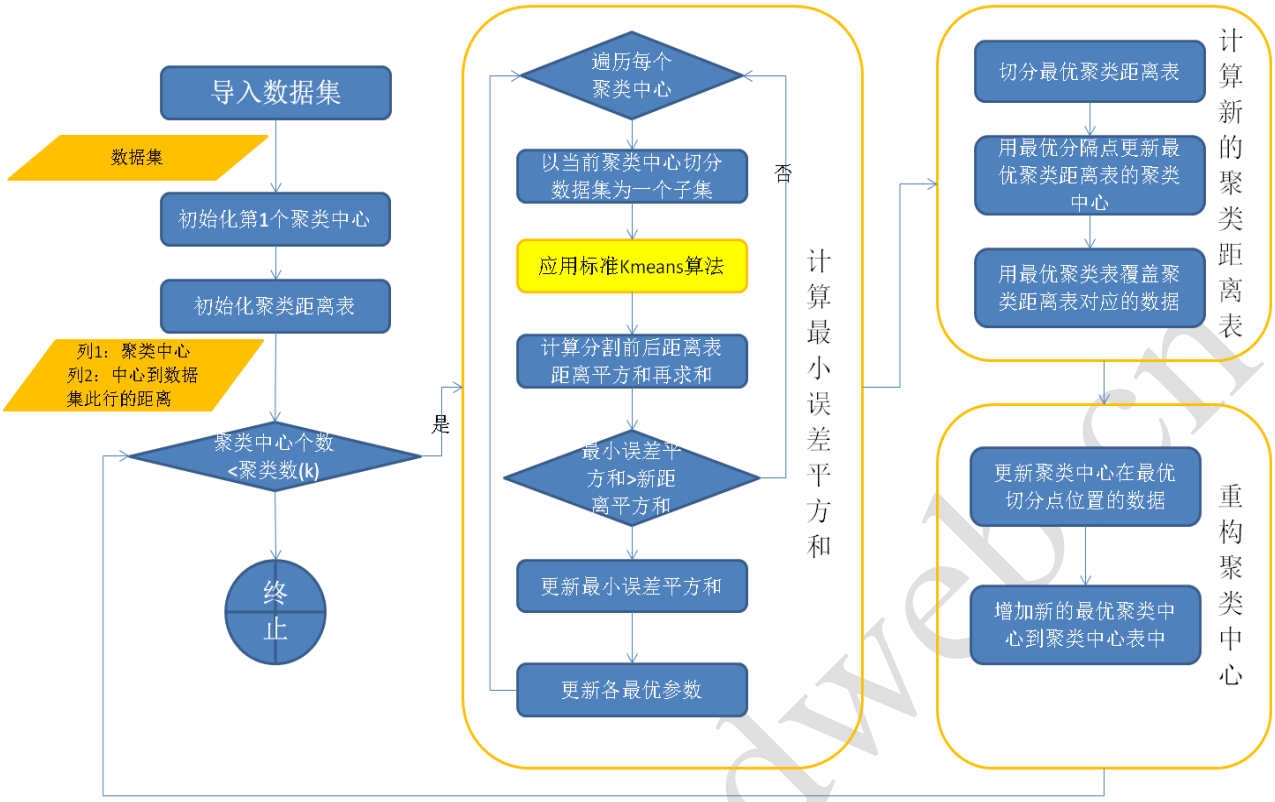
3）对异常偏离的数据敏感；

本文针对典型的 K-means 算法这三个方面的缺陷进行改进。由于传统的 K-means 算法聚类结果受到初始聚类中心点选择的影响，因此在传统的 K-means 算法的基础上进行改进算法对初始中心点选取比较严格，各中心点的距离较远，这就避免了初始聚类中心会选到一个类上，一定程度上克服了算法限入局部最优状态。算法对以上三点进行，得到的二分 K-means 算法更有效。

二分 k 均值（bisecting k-means）算法的主要思想是：首先将所有点作为一个簇，然后将该簇一分为二。之后选择能最大程度降低聚类代价函数（也就是误差平方和）的簇划分为两个簇。以此进行下去，直到簇的数目等于用户给定的数目 k 为止。

以上隐含着一个原则是：因为聚类的误差平方和能够衡量聚类性能，该值越小表示数据点月接近于它们的质心，聚类效果就越好。所以我们就需要对误差平方和最大的簇进行再一次的划分，因为误差平方和越大，表示该簇聚类越不好，越有可能是多个簇被当成一个簇了，所以我们首先需要对这个簇进行划分。

二分 Kmeans 算法流程



二分 Kmeans 算法

```
# -*- coding: utf-8 -*-
# Filename : 03bikMeans1.py

from numpy import *
import numpy as np
import operator
import kMeans2
import matplotlib.pyplot as plt

# 从文件构建的数据集
dataSet = kMeans2.loadDataSet("testSet.txt")
dataSet = mat(dataSet) # 转换为矩阵形式

# 绘制 dataSet 散点图形
for mydata in dataSet:
    plt.scatter(mydata[0,0],mydata[0,1],c='blue',marker='o')
```

```
k = 4      #聚类数
m = shape(dataSet)[0] #dataSet 的行数

#  聚类距离表:行数与数据集相同
#  列 1：数据集对应的聚类中心,列 2:数据集行向量到聚类中心的距离
clusterAssment = mat(zeros((m,2)))

#  初始化第一个聚类中心: dataSet 的每一列的均值
centroid0 = mean(dataSet, axis=0).tolist()[0]
#  把这个随机聚类中心加入聚类中心列表中
centList =[centroid0]

#  初始化聚类距离表,距离方差: sum((centroid0-dataSet[j,:])^2)
# clusterAssment[j,1] = [0,方差]
# print centroid0
for j in range(m):
    clusterAssment[j,1] = sum(power(mat(centroid0)- dataSet[j,:], 2))
# print clusterAssment

#  判断  centList 的长度是否小于 k
while (len(centList) < k):
    lowestSSE = inf #  初始化最小误差平方和为无穷大。核心参数，这个值越小就说明聚类的效果越好。
    #  遍历 cenList 的每个向量
    #----1.  使用 clusterAssment 计算 lowestSSE，以此确定:bestCentToSplit、bestNewCents、bestClustAss----#
    for i in range(len(centList)):
        # nonzero(clusterAssment[:,0].A==i):返回 clusterAssment 第 1 列==i 所构成的行下标数组
        cAIndx = nonzero(clusterAssment[:,0].A==i)[0]
        #  用 cAIndx 过滤数据集,构成一个子集
        ptsInCurrCluster = dataSet[cAIndx,:]
        #  应用标准 kMeans 算法(k=2),将 ptsInCurrCluster 划分出两个聚类中心,以及对应的聚类距离表
        centroidMat,splitClustAss = kMeans2.kMeans(ptsInCurrCluster, 2)
        #  计算 splitClustAss 的距离平方和
        sseSplit = sum(splitClustAss[:,1])
        #  返回 clusterAssment 第 1 列!=i 所构成的行下标数组
        ucAIndx = nonzero(clusterAssment[:,0].A!=i)[0]
        #  计算 clusterAssment[clusterAssment 第 1 列!=i 的距离平方和
        sseNotSplit = sum(clusterAssment[ucAIndx,1])
        #  算法公式: lowestSSE = sseSplit + sseNotSplit
        if (sseSplit + sseNotSplit) < lowestSSE:
            bestCentToSplit = i                          #  确定聚类中心的最优分隔点
            # print "i:",i
            bestNewCents = centroidMat                   #  用新的聚类中心更新最优聚类中心
            bestClustAss = splitClustAss.copy()          #  深拷贝聚类距离表为最优聚类距离表
```

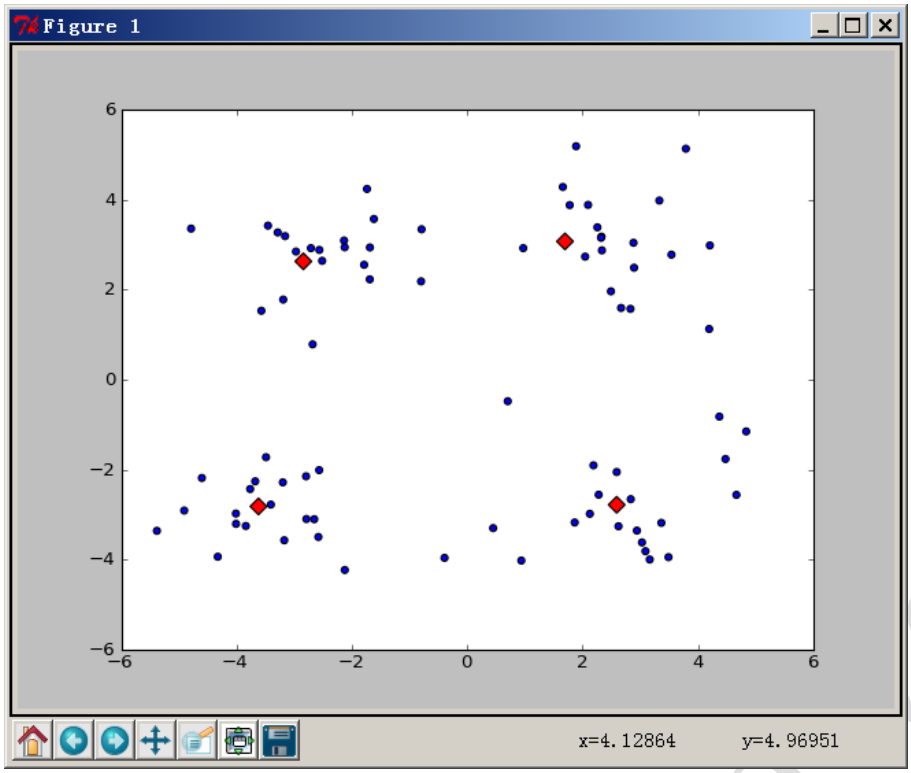
```
                # print "bestClustAss:",bestClustAss
                lowestSSE = sseSplit + sseNotSplit    # 更新 lowestSSE
                # print "lowestSSE:",lowestSSE
        # 回到外循环
        #----2. 计算新的 clusterAssment----#
        # 返回 bestClustAss 第 1 列==1 所构成的行下标数组
        bIndx0 = nonzero(bestClustAss[:,0].A == 1)[0] # 分了两部分，第一部分
        # 为 bestClustAss[bIndx0,0]赋值为聚类中心的索引
        bestClustAss[bIndx0,0] = len(centList)
        # 返回 bestClustAss 第 1 列==0 所构成的行下标数组
        bIndx1 = nonzero(bestClustAss[:,0].A == 0)[0] # 分了两部分，第二部分
        # 用最优分隔点的取值填充 bestClustAss 第 1 列值==0 的列向量
        bestClustAss[bIndx1,0] = bestCentToSplit
        # print "bestClustAss:",bestClustAss
        #以上为计算 bestClustAss
        # 返回 clusterAssment 第 1 列值==bestCentToSplit 所构成的行下标数组
        cIndxb = nonzero(clusterAssment[:,0].A == bestCentToSplit)[0]
        # 更新 clusterAssment 对应最优分隔点下标的距离，使距离值等于最优聚类距离对应的值
        clusterAssment[cIndxb,:]= bestClustAss
        #以上为计算 clusterAssment

        #----3. 用最优分隔点来重构聚类中心----#
        # 覆盖: bestNewCents[0,:].tolist()[0]附加到原有聚类中心的 bestCentToSplit 位置
        # 增加: 聚类中心增加一个新的 bestNewCents[1,:].tolist()[0]向量
        centList[bestCentToSplit] = bestNewCents[0,:].tolist()[0]
        centList.append(bestNewCents[1,:].tolist()[0])
        # print "centList:",centList
        # 以上为计算 centList

print "cenList:",mat(centList)
print "clusterAssment:", clusterAssment
# 绘制聚类中心图形
for cent in centList:
        plt.scatter(cent[0],cent[1],s=60,c='red',marker='D')
plt.show()
```

输出:

```
cenList: [[ 2.80293085 -2.7315146 ]       # 稳定的聚类中心
 [ 2.6265299    3.10868015]
 [-3.38237045 -2.9473363 ]
 [-2.46154315   2.78737555]]
clusterAssment:                           # 聚类距离表
 [[    1.00000000e+00     2.32019150e+00]
```

```
[   3.00000000e+00    1.39004893e+00]
[   0.00000000e+00    6.63839104e+00]
[   2.00000000e+00    4.16140951e+00]
[   1.00000000e+00    2.76967820e+00]
[   3.00000000e+00    2.80101213e+00]
[   0.00000000e+00    5.85909807e+00]
[   2.00000000e+00    1.50646425e+00]
[   1.00000000e+00    2.29348924e+00]
[   3.00000000e+00    6.45967483e-01]
[   0.00000000e+00    1.74010499e+00]
[   2.00000000e+00    3.77769471e-01]
[   1.00000000e+00    2.51695402e+00]
[   3.00000000e+00    1.38716420e-01]
[   0.00000000e+00    9.47633071e+00]
[   2.00000000e+00    9.97310599e+00]
[   1.00000000e+00    2.39726914e+00]
[   3.00000000e+00    3.10242360e+00]
[   0.00000000e+00    4.11084375e-01]
[   2.00000000e+00    4.74890795e-01]
[   1.00000000e+00    1.38706133e-01]
[   3.00000000e+00    5.10240996e-01]
[   0.00000000e+00    1.05700176e+00]
[   2.00000000e+00    2.90181828e-02]
[   1.00000000e+00    1.31601105e+00]
[   3.00000000e+00    9.08203769e-01]
[   0.00000000e+00    5.02608557e-01]
[   2.00000000e+00    4.57942717e-01]
[   1.00000000e+00    2.13786618e-01]
[   3.00000000e+00    4.05632356e+00]
[   0.00000000e+00    5.14171888e+00]
[   2.00000000e+00    5.56237495e-01]
[   1.00000000e+00    4.76142736e-01]
[   3.00000000e+00    1.54414110e+00]
[   0.00000000e+00    6.10930460e+00]
[   2.00000000e+00    9.47660177e-01]
[   1.00000000e+00    4.87745774e+00]
[   3.00000000e+00    3.12703929e+00]
[   0.00000000e+00    6.45118831e-03]
[   2.00000000e+00    3.01415411e-01]
[   1.00000000e+00    8.84955695e-01]
[   3.00000000e+00    7.98870968e-02]
[   0.00000000e+00    5.23673430e-01]
[   2.00000000e+00    3.24171404e+00]
```

```
[   1.00000000e+00   9.32523506e-02]
[   3.00000000e+00   9.13705455e-01]
[   0.00000000e+00   1.25766593e+00]
[   2.00000000e+00   4.09563895e-01]
[   1.00000000e+00   9.46987842e-01]
[   3.00000000e+00   2.63836399e+00]
[   0.00000000e+00   5.20371222e-01]
[   2.00000000e+00   1.86796790e+00]
[   1.00000000e+00   5.46768776e+00]
[   3.00000000e+00   5.73153563e+00]
[   0.00000000e+00   3.12040332e-01]
[   2.00000000e+00   3.93986735e-01]
[   1.00000000e+00   1.32864695e+00]
[   3.00000000e+00   2.38032454e-02]
[   0.00000000e+00   1.07872914e+00]
[   2.00000000e+00   4.35369355e-01]
[   1.00000000e+00   4.55502856e-01]
[   3.00000000e+00   1.96212809e-02]
[   0.00000000e+00   1.95213538e+00]
[   2.00000000e+00   1.54154401e+00]
[   1.00000000e+00   1.26364010e+00]
[   3.00000000e+00   1.33108375e+00]
[   0.00000000e+00   3.02422139e-01]
[   2.00000000e+00   5.58860689e-01]
[   1.00000000e+00   9.52516316e-02]
[   3.00000000e+00   6.25129762e-01]
[   0.00000000e+00   8.41875177e-01]
[   2.00000000e+00   2.06159470e+00]
[   1.00000000e+00   6.39227291e+00]
[   3.00000000e+00   2.01200372e-01]
[   0.00000000e+00   3.51030769e+00]
[   2.00000000e+00   9.83287604e-01]
[   1.00000000e+00   7.06014703e-02]
[   3.00000000e+00   2.59901305e-01]
[   0.00000000e+00   3.74491207e+00]
[   2.00000000e+00   2.32143993e+00]]
```



书中实例（略）