菊安酱的机器学习第9期

菊安酱的直播间：https://live.bilibili.com/14988341

每周一晚8:00 菊安酱和你不见不散哦~(^o^)/~

更新日期：2018-12-28

作者：菊安酱

课件内容说明：

- 本文为作者原创，转载请注明作者和出处
- 如果想获得此课件及录播视频，可扫描左边二维码，回复"k"进群
- 如果想获得2小时完整版视频，可扫描右边二维码或点击如下链接
- 若有任何疑问，请给作者留言。

交流群二维码

完整版视频及课件

直播视频及课件：http://www.peixun.net/view/1278.html

完整版视频及课件：http://edu.cda.cn/course/966

12期完整版课纲

直播时间：每周一晚8:00

直播内容：

时间	期数	算法
2018/11/05	第1期	k-近邻算法
2018/11/12	第2期	决策树
2018/11/19	第3期	朴素贝叶斯
2018/11/26	第4期	Logistic回归
2018/12/03	第5期	支持向量机
2018/12/10	第6期	AdaBoost 算法
2018/12/17	第7期	线性回归
2018/12/24	第8期	树回归
2018/12/28	第9期	K-均值聚类算法
2019/01/07	第10期	Apriori 算法
2019/01/14	第11期	FP-growth 算法
2019/01/21	第12期	奇异值分解SVD

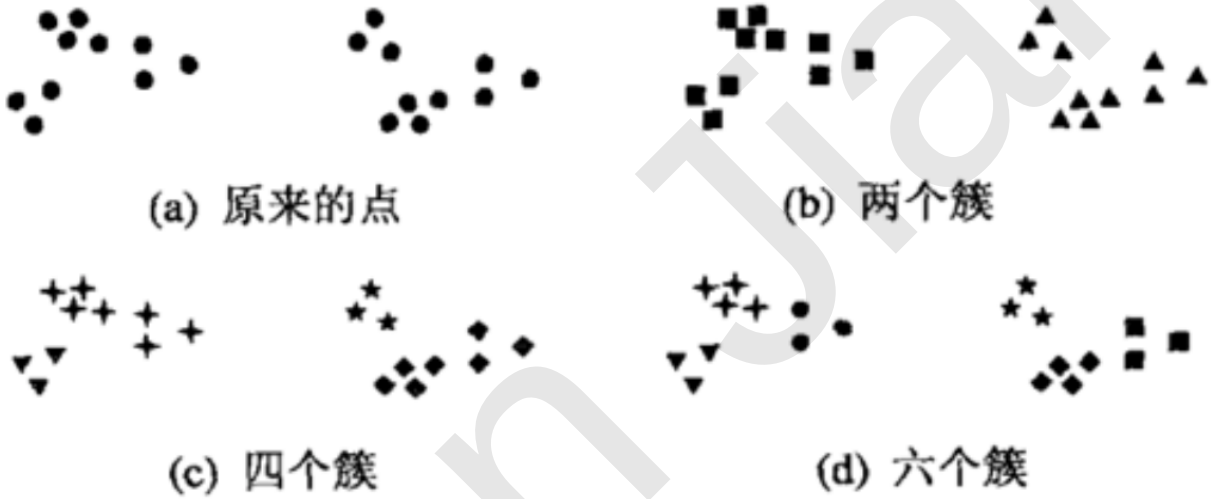
k-均值聚类算法

一、聚类分析概述

聚类分析是无监督类机器学习算法中最常用的一类，其目的是将数据划分成有意义或有用的组（也被称为**簇**）。组内的对象相互之间是相似的（相关的），而不同组中的对象是不同的（不相关的）。组内的相似性 （同质性）越大，组间差别越大，聚类就越好。

1. 簇的定义

简单来说，簇就是分类结果中的类，但实际上簇并没有明确的定义，并且簇的划分没有客观标准，我们可以利用下图来理解什么是簇。该图显示了 20个点和将它们划分成簇的3种不同方法。标记的形状指示簇的隶属关系。下图分别将数据划分成两部分、四部分和六部分。将2个较大的簇每一个都划分成3个子簇可能是人的视觉系统造成的假象。此外，说这些点形成4个簇可能也不无道理。该图表明簇的定义是不精确的，而最好的定义依赖于数据的特性和期望的结果。



2.常用的聚类算法

最常用的聚类算法有以下三种：

•**K-means聚类**：也称为**K均值聚类**，它试图发现k（用户指定个数）个不同的簇，并且每个簇的中心采用簇中所含值的均值计算而成。

•**层次聚类**：层次聚类(hierarchical clustering)试图在不同层次对数据集进行划分，从而形成树形的聚类结构。

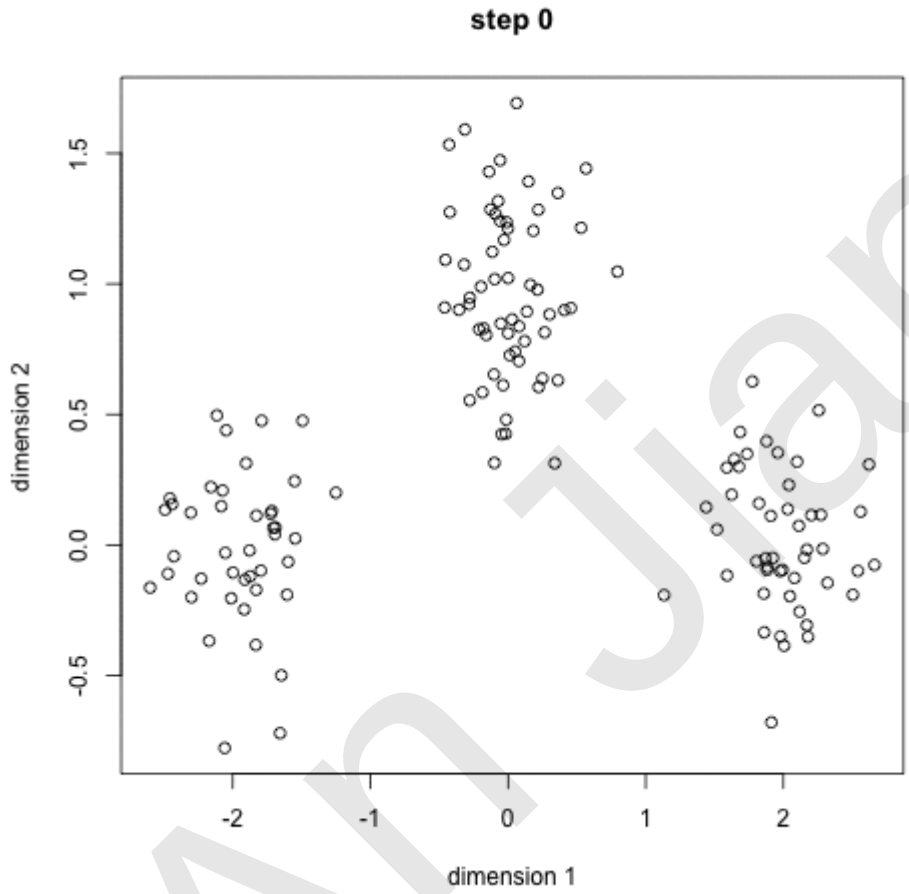
•**DBSCAN**：这是一种基于密度的聚类算法，簇的个数由算法自动地确定。低密度区域中的点被视为噪声而忽略，因此DBSCAN不产生完全聚类。

二、K-均值聚类算法

K均值是发现给定数据集的k个簇的算法。簇个数k是用户给定的，每一个簇通过其质心（centroid）来描述。

K均值工作流程是这样的：首先，随机选择K个初始质心，其中K是用户指定的参数，即所期望的簇的个数。然后将数据集中每个点指派到最近的质心，而指派到一个质心的点即为一个簇。然后，根据指派到簇的点，将每个簇的质心更新为该簇所有点的平均值。重复指派和更新步骤，直到簇不发生变化，或等价地，直到质心不发生变化。

该过程如下图所示，该图显示如何从3个质心出发，通过12次指派和更新，找出最后的簇。质心用符号“x”指示；属于同一个簇的所有点具有相同颜色的标记。



1. K-均值算法的python实现

根据k-均值算法的工作流程，我们可以写出伪代码：

```
创建k个点作为初始质心（通常是随机选择）
当任意一个点的簇分配结果发生改变时：
    对数据集中的每个点：
        对每个质心：
            计算质心与数据点之间的距离
        将数据点分配到据其最近的簇
    对每个簇，计算簇中所有点的均值并将均值作为新的质心
直到簇不再发生变化或者达到最大迭代次数
```

在伪代码中，有提到“最近”的说法，那就意味着要进行某种距离的计算。**附录1**中给大家总结了几种距离度量方法。这里我们使用的是最简单的欧式距离。

1.1 导入数据集

此处先以经典的鸢尾花数据集为例，来帮助我们建模，数据存放在iris.txt中

```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt

#导入数据集
iris = pd.read_csv('iris.txt',header = None)
iris.head()
iris.shape
```

1.2 构建距离计算函数

我们需要定义一个两个长度相等的数组之间欧式距离计算函数，在不直接应用计算距离计算结果，只比较距离远近的情况下，我们可以用距离平方和代替距离进行比较，化简开平方运算，从而减少函数计算量。

此外需要说明的是，涉及到距离计算的，一定要注意量纲的统一。如果量纲不统一的话，模型极易偏向量纲大的那一方。**附录2**给大家总结了一些归一化的方法，供大家参考。此处选用鸢尾花数据集，基本不需要考虑量纲问题。

```
"""
函数功能：计算两个数据集之间的欧式距离
输入：两个array数据集
返回：两个数据集之间的欧氏距离（此处用距离平方和代替距离）
"""
def distEclud(arrA, arrB):
    d = arrA - arrB
    dist = np.sum(np.power(d, 2), axis=1)
    return dist
```

1.3 编写自动生成随机质心的函数

在定义随机质心生成函数时，首先需要计算每列数值的范围，然后从该范围中随机生成指定个数的质心。此处我们使用numpy.random.uniform()函数生成随机质心。

```
"""
函数功能：随机生成k个质心
参数说明：
    dataSet:包含标签的数据集
    k：簇的个数
返回：
    data_cent：K个质心
"""
def randCent(dataSet, k):
    n = dataSet.shape[1]
    data_min = dataSet.iloc[:, :n-1].min()
    data_max = dataSet.iloc[:, :n-1].max()
    data_cent = np.random.uniform(data_min,data_max,(k, n-1))
```

```
    return data_cent
```

验证上述定义函数，在iris中随机生成三个质心

```
iris_cent = randCent(iris, 3)
iris_cent
```

1.4 编写K-Means聚类函数

在执行K-Means的时候，需要不断的迭代质心，因此我们需要两个可迭代容器来完成该目标：

第一个容器用于存放和更新质心，该容器可考虑使用list来执行，list不仅是可迭代对象，同时list内不同元素索引位置也可用于标记和区分各质心，即各簇的编号；

第二个容器则需要记录、保存和更新各点到质心之间的距离，并能够方便对其进行比较，该容器考虑使用一个三列的数组来执行，其中第一列用于存放最近一次计算完成后某点到各质心的最短距离，第二列用于记录最近一次计算完成后根据最短距离得到的代表对应质心的数值索引，即所属簇，即质心的编号，第三列用于存放上一次某点所对应质心编号（某点所属簇），后两列用于比较质心发生变化后某点所属簇的情况是否发生变化。

```
"""
函数功能：k-均值聚类算法
参数说明：
    dataSet：带标签数据集
    k:簇的个数
    distMeas:距离计算函数
    createCent:随机质心生成函数
返回：
    centroids:质心
    result_set:所有数据划分结果
"""
def kMeans(dataSet, k, distMeas=distEclud, createCent=randCent):
    m,n = dataSet.shape
    centroids = createCent(dataSet, k)
    clusterAssment = np.zeros((m,3))
    clusterAssment[:, 0] = np.inf
    clusterAssment[:, 1: 3] = -1
    result_set = pd.concat([dataSet, pd.DataFrame(clusterAssment)], axis=1,
ignore_index = True)
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        for i in range(m):
            dist = distMeas(dataSet.iloc[i, :n-1].values, centroids)
            result_set.iloc[i, n] = dist.min()
            result_set.iloc[i, n+1] = np.where(dist == dist.min())[0]
        clusterChanged = not (result_set.iloc[:, -1] == result_set.iloc[:, -2]).all()
        if clusterChanged:
            cent_df = result_set.groupby(n+1).mean()
            centroids = cent_df.iloc[:,:n-1].values
            result_set.iloc[:, -1] = result_set.iloc[:, -2]
    return centroids, result_set
```

鸢尾花数据集带进去，查看模型运行效果：

```
iris_cent,iris_result = kMeans(iris, 3)
iris_cent
iris_result.head()
```

有以下几点需要特别注意：

- 设置统一的操作对象result_set

  为了调用和使用的方便，此处将clusterAssment容易转换为了DataFrame并与输入DataFrame合并，组成的对象可作为后续调用的统一对象，该对象内即保存了原始数据，也保存了迭代运算的中间结果，包括数据所属簇标记和数据质心距离等，该对象同时也作为最终函数的返回结果；

- 判断质心发生是否发生改变条件

  注意，在K-Means中判断质心是否发生改变，即判断是否继续进行下一步迭代的依据并不是某点距离新的质心距离变短，而是某点新的距离向量（到各质心的距离）中最短的分量位置是否发生变化，即质心变化后某点是否应归属另外的簇。在质心变化导致各点所属簇发生变化的过程中，点到质心的距离不一定会变短，即判断条件不能用下述语句表示

  ```
  if not (result_set.iloc[:, -1] == result_set.iloc[:, -2]).all()
  ```

- 合并DataFrame后索引值为n的列

  这里有个小技巧，能够帮助迅速定位DataFrame合并后列的索引，即两个DF合并后后者的第一列在合并后的DF索引值为n，第二列索引值为n+1

- 质心和类别一一对应

  即在最后生成的结果中，centroids的行标即为result_set中各点所属类别

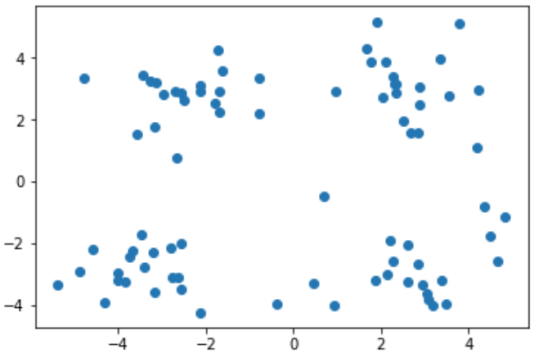
2. 算法验证

数编写完成后，先以testSet数据集测试模型运行效果（为了可以直观看出聚类效果，此处采用一个二维数据集进行验证）。testSet数据集是一个二维数据集，每个观测值都只有两个特征，且数据之间采用空格进行分隔，因此可采用pd.read_table()函数进行读取

```
testSet = pd.read_table('testSet.txt', header=None)
testSet.head()
testSet.shape
```

然后利用二维平面图形观察其分布情况

```
plt.scatter(testSet.iloc[:,0], testSet.iloc[:,1]);
```



可以大概看出数据大概分布在空间的四个角上，后续我们将对此进行验证。然后利用我们刚才编写的K-Means算法对其进行聚类，在执行算法之前需要添加一列虚拟标签列（算法是从倒数第二列开始计算特征值）

```
label = pd.DataFrame(np.zeros(testSet.shape[0]).reshape(-1, 1))
test_set = pd.concat([testSet, label], axis=1, ignore_index = True)
test_set.head()
```

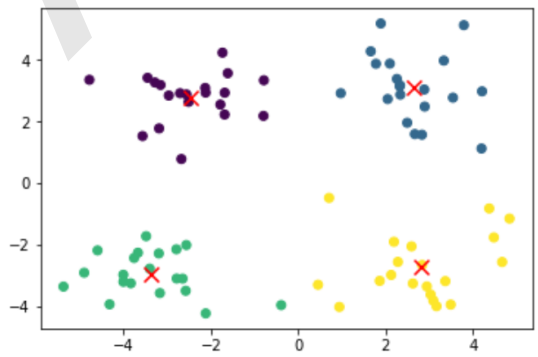
然后带入算法进行计算，根据二维平面坐标点的分布特征，我们可考虑设置四个质心，即将其分为四个簇，并简单查看运算结果

```
test_cent, test_cluster = kMeans(test_set, 4)
test_cent
test_cluster.head()
```

将分类结果进行可视化展示，使用scatter函数绘制不同分类点不同颜色的散点图，同时将质心也放入同一张图中进行观察

```
plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1], c=test_cluster.iloc[:,
-1])
plt.scatter(test_cent[:, 0], test_cent[:, 1], color='red',marker='x',s=100);
```

能够看出，K-Means模型判别结果和我们初步判别结果类似，二维空间内偏向于分布在四个角上。



3.误差平方和SSE计算

误差平方和（SSE）是聚类算法模型最重要评估指标，接下来，在手动编写的K-Means快速聚类函数基础上计算结果集中的误差平方和。在kMeans函数生成的结果result_set中，第n列，也就是我们定义的clusterAssment容器中的第一列就保留了最近一次分类结果的某点到对应所属簇质心的距离平方和，因此我们只需要对result_set中的第n列进行简单求和汇总，即可得聚类模型误差平方和。

```
test_cluster.iloc[:,3].sum()
```

```
iris_result.iloc[:,5].sum()
```

```
In [17]: test_cluster.iloc[:, 3].sum()
Out[17]: 149.95430467642635
In [18]: iris_result.iloc[:,5].sum()
Out[18]: 78.94084142614602
```

当然，对于聚类算法而言，误差平方和仍然有一定的局限性，主要体现在以下几点：

- 对于任意数据集而言，聚类误差平方和和质心数量高度相关，随着质心增加误差平方和将逐渐下降，虽然下降过程偶尔会有小幅起伏，不是严格递减，极限情况是质心数量和数据集行数保持一致，此时误差平方和将趋于零（思考为何不为0）；
- 同时，误差平方和还与数据集本身数据量大小、量纲大小、数据维度高度相关，数据量越大、量纲越大、维度越高则在相同质心数量情况下误差平方和也将更大；

因此，模型误差平方和没有绝对意义，比较不同数据集聚类结果的误差平方和没有任何意义，误差平方和在聚类分析中主要作用有以下两点：

- 确定模型最优化目标，结合距离计算方法进而推导质心选取方法；
- 对于确定数据集可绘制横轴为质心数量、纵轴为误差平方和的曲线，可以判断，曲线整体将呈现下降趋势，我们可从中找到“骤然下降”的某个拐点，则该点可作为聚类分类数量的参考值，即可利用SSE绘制聚类数量学习曲线。当然，由于聚类分析本身属于探索性质的算法，曲线拐点给出的值只是当前数据维度和数据量的情况，数据在高维空间内的分布可能呈现的集中分布趋势，并不一定代表客观事物的自然规律，也不是所有数据集的聚类分类数量学习曲线都存在拐点；

接下来，尝试绘制聚类分类数量的学习曲线，这里仍然考虑自定义一个函数来进行绘制，此处需要注意，质心数量选取建议从2开始，质心为1的时候SSE数值过大，对后续曲线显示效果有较大影响

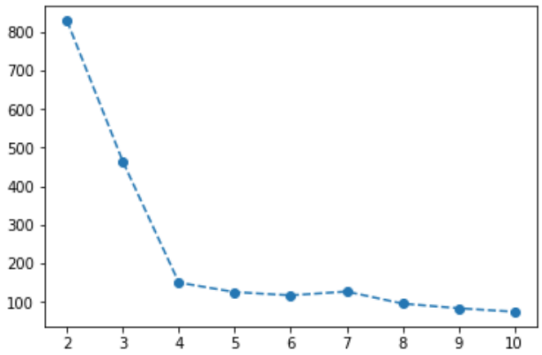
```
"""
函数功能：聚类学习曲线
参数说明：
    dataSet：原始数据集
    cluster：Kmeans聚类方法
    k：簇的个数
返回：误差平方和SSE
"""
def kcLearningCurve(dataSet, cluster = kMeans, k=10):
    n = dataSet.shape[1]
    SSE = []
    for i in range(1, k):
        centroids, result_set = cluster(dataSet, i+1)
        SSE.append(result_set.iloc[:,n].sum())
```

```
    plt.plot(range(2, k+1), SSE, '--o')
    return SSE
```

然后在已有数据集上进行测试

```
kcLearningCurve(test_set)
```

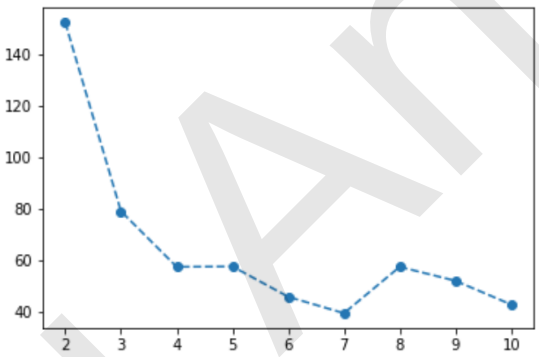
```
In [21]: kcLearningCurve(test_set);
```



此时能看出，质心由2增加到4的时候SSE下降速度较快，而从4开始，随着质心增加SSE下降速度有所递减，因此当质心为4的时候聚类效果较好

```
kcLearningCurve(iris)
```

```
In [25]: kcLearningCurve(iris);
```



而在iris数据集中，质心选取3个或4个为佳，其中质心选取4个时的聚类效果比选取3个质心要更好。这里虽然生物学上数据集对象本身包含了三个不同类别的鸢尾花，但就采集的数据和字段而言，从数据高维空间分布来说更加倾向于可分为4个簇，从侧面也说明数据量和数据集特征将影响模型判别效果，换而言之就是数据本身将影响模型对数据背后客观规律的探索。

三、模型收敛稳定性探讨

在执行前面聚类算法的过程中，好像虽然初始质心是随机生成的，但最终分类结果均保持一致。若初始化参数是随机设置（如此处初始质心位置是随机生成的），但最终模型在多次迭代之后稳定输出结果，则可认为最终模型收敛，所谓迭代是指上次运算结果作为下一次运算的条件参与计算，kMeans的每次循环本质上都是迭代，我们利用上次生成的中心点参与到下次距离计算中。

同时，在之前我们编写kMeans聚类算法中我们设置的收敛条件比较简单，就是最近两次迭代各点归属簇的划分结果，若结果不发生变化，则表明结果收敛，停止迭代。但实际上这种收敛条件并不稳定，尤其是在我们采用均值作为质心带入计算的过程中，实际上是采用了梯度下降作为优化手段，但梯度下降本质上是无约束条件下局部最优手段，局部最优手段最终不一定导致全局最优，即我们使用均值作为质心带入迭代，最终依据收敛判别结果计算的最终结果一定是基于初始质心的局部最优结果，但不一定是全局最优的结果，因此其实刚才的结果并不稳定，最终分类和计算的结果会受到初始质心选取的影响。

接下来验证初始质心选取最终如何影响K-means聚类结果的。这里我们可提前设置随机数种子，由于我们的初始质心由np.random类函数生成，所以随机数种子也要用np.random类方法生成

```
np.random.seed(123)
```

然后执行算法，注意，这里我们每执行一次kMeans函数，初始质心就会随机生成一次，我们需要观察的是重复执行算法过程会不会最终输出不同的分类结果。为方便更直观的表达，此处仅以test_set数据集验证过程为例进行讨论

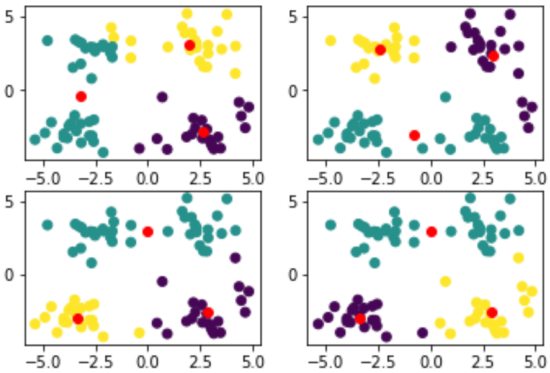
- 验证test_set数据集三分类结果

  同时，利用多子图功能进行可视化结果展示

```
np.random.seed(123)
for i in range(1, 5):
    plt.subplot(2, 2, i)
    test_cent, test_cluster = kMeans(test_set, 3)
    plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1],
c=test_cluster.iloc[:, -1])
    plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red')
    print(test_cluster.iloc[:, 3].sum())
```

In [23]:
```
np.random.seed(123)
for i in range(1, 5):
    plt.subplot(2, 2, i)
    test_cent, test_cluster = kMeans(test_set, 3)
    plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1], c=test_cluster.iloc[:, -1])
    plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red')
    print(test_cluster.iloc[:, 3].sum())
```

```
463.6496809222772
506.0588518418539
405.13810196190366
405.13810196190366
```

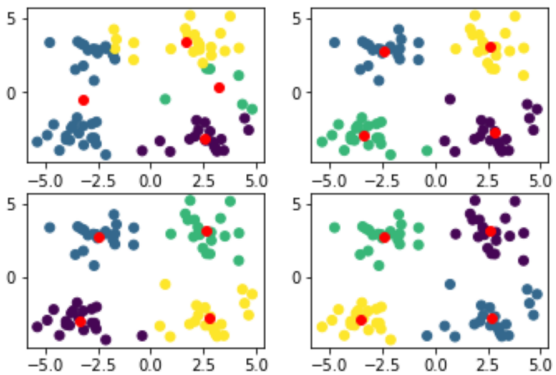


- 验证test_set数据集四分类结果

```
np.random.seed(123)
for i in range(1, 5):
    plt.subplot(2, 2, i)
    test_cent, test_cluster = kMeans(test_set, 4)
    plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1],
c=test_cluster.iloc[:, -1])
    plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red')
    print(test_cluster.iloc[:, 3].sum())
```

In [24]:
```
np.random.seed(123)
for i in range(1, 5):
    plt.subplot(2, 2, i)
    test_cent, test_cluster = kMeans(test_set, 4)
    plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1], c=test_cluster.iloc[:, -1])
    plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red')
    print(test_cluster.iloc[:, 3].sum())
```

```
438.99925511275205
149.95430467642635
149.95430467642635
150.62604907269227
```

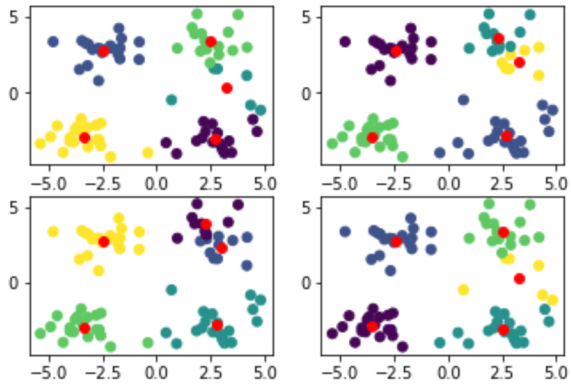


- 验证test_set数据集五分类结果

```
np.random.seed(123)
for i in range(1, 5):
    plt.subplot(2, 2, i)
    test_cent, test_cluster = kMeans(test_set, 5)
    plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1],
c=test_cluster.iloc[:, -1])
    plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red')
    print(test_cluster.iloc[:, 3].sum())
```

In [25]:
```
np.random.seed(123)
for i in range(1, 5):
    plt.subplot(2, 2, i)
    test_cent, test_cluster = kMeans(test_set, 5)
    plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1], c=test_cluster.iloc[:, -1])
    plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red')
    print(test_cluster.iloc[:, 3].sum())
```

```
132.51859467191917
134.72812560240237
134.4631443966685
131.89160568812764
```



大家可以继续对iris数据集和更多分类的test_set数据集进行测试，最终我们能得出以下结论：

- 初始质心的随机选取在kMeans中将最终影响聚类结果；
- 质心数量从某种程度上将影响这种随机影响的程度，如果质心数量的选取和数据的空间集中分布情况越相类似，则初始质心的随机选取对聚类结果可能造成影响的概率越小，当然，这也和质心随机生成的随机方法也高度相关。

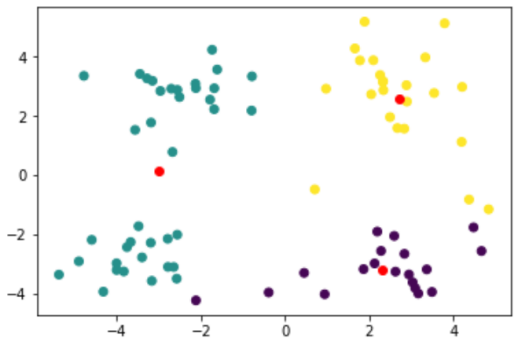
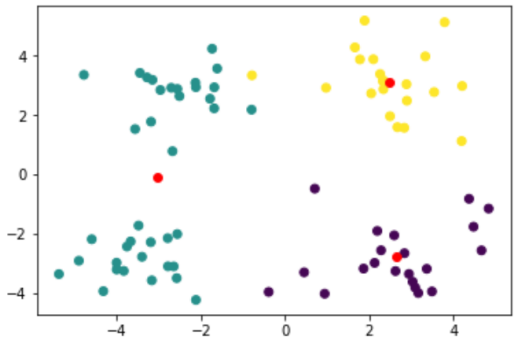
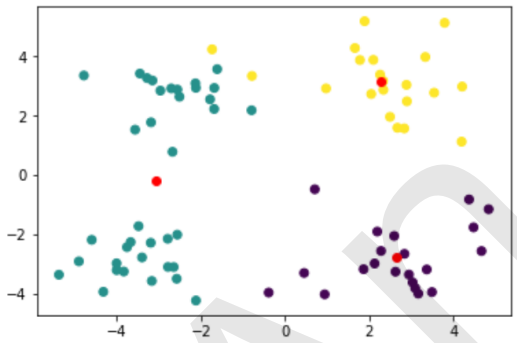
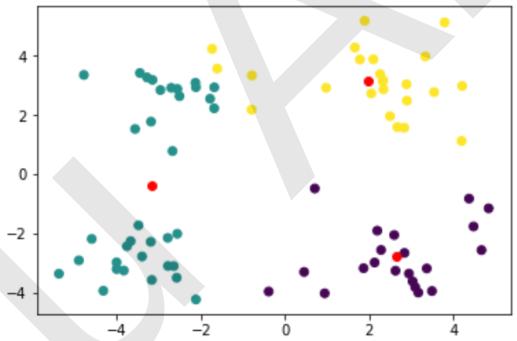
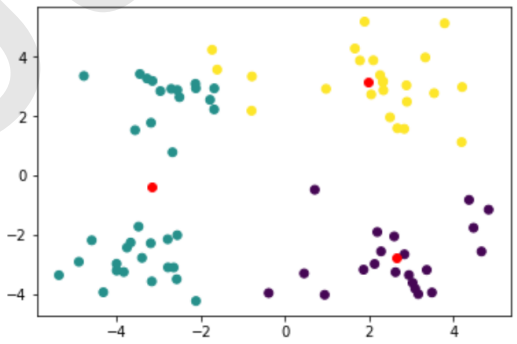
接下来，我们仍以test_set三分类为例，稍微修改kMeans函数，使其每一步都生成一张点图，借以查看随机初始化质心是如何一步步最终影响聚类结果的

```
"""
函数功能：绘制迭代聚类点图
参数说明：
    dataSet：原始数据集
    k：簇个数k
    distMeas：距离计算函数
    createCent：随机质心生成函数
返回：每一步生成的聚类点图
"""
def kMeans_1(dataSet, k, distMeas=distEclud, createCent=randCent):
    m,n = dataSet.shape
    centroids = createCent(dataSet, k)
    clusterAssment = np.zeros((m,3))
    clusterAssment[:, 0] = np.inf
    clusterAssment[:, 1: 3] = -1
    result_set = pd.concat([dataSet, pd.DataFrame(clusterAssment)], axis=1,
ignore_index = True)
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        for i in range(m):
            dist = distMeas(dataSet.iloc[i, :n-1].values, centroids)
```

```
            result_set.iloc[i, n] = dist.min()
            result_set.iloc[i, n+1] = np.where(dist == dist.min())[0]
        clusterChanged = not (result_set.iloc[:, -1] == result_set.iloc[:, -2]).all()
        if clusterChanged:
            cent_df = result_set.groupby(n+1).mean()
            centroids = cent_df.iloc[:,:n-1].values
            result_set.iloc[:, -1] = result_set.iloc[:, -2]
        plt.scatter(result_set.iloc[:,0], result_set.iloc[:, 1], c=result_set.iloc[:,
-1])
        plt.plot(centroids[:, 0], centroids[:, 1], 'o', color='red')
        plt.show()
```

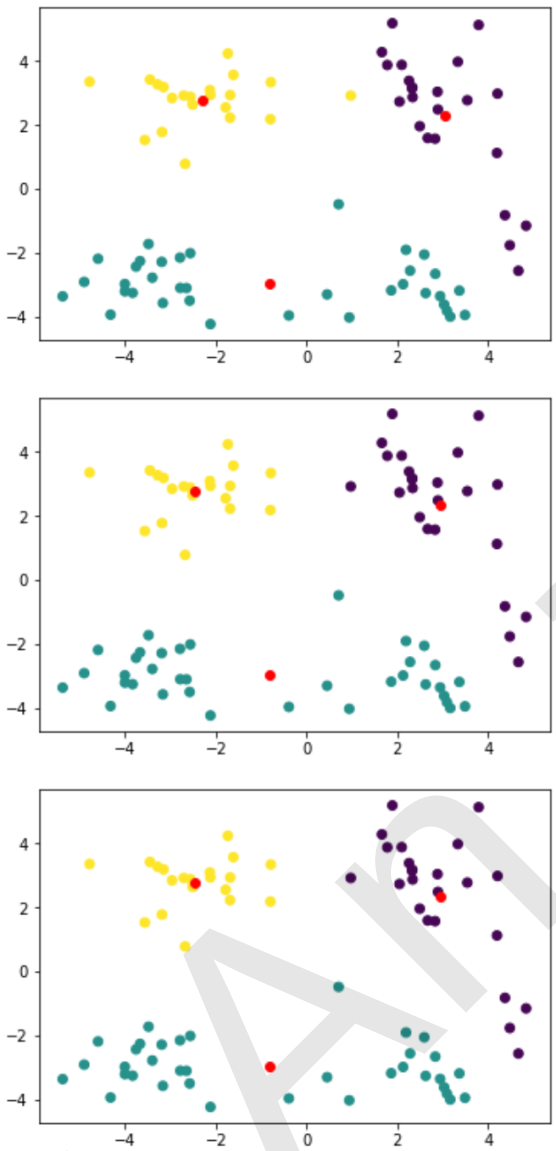
```
np.random.seed(123)
kMeans_1(test_set, 3)
```

In [30]:
```
np.random.seed(123)
kMeans_1(test_set, 3)
```

```
kMeans_1(test_set, 3)
```

```
In [31]: kMeans_1(test_set, 3)
```



从中我们可清楚地看到随机初始质心是如何影响最终聚类分类结果的。解决该问题的办法，即尽量降低初始化质心随机性对最后聚类结果造成影响的方案有以下几种：

- 在设始化质心随机生成的过程中，尽可能的让质心更加分散

  这点其实我们在利用np.random.random进行[0,1)区间内取均匀分布抽样而不是进行正态分布抽样，就是为了能够尽可能的让初始质心分散；

- 人工制定初始质心

  即在观察数据集分布情况后，手工设置初始质心，此举也能降低随机性影响，但需要人为干预；

- 增量的更新质心

  可以在点到簇的每次指派之后，增量地更新质心，而不是在所有的点都指派到簇中之后才更新簇质心。注意，每步需要零次或两次簇质心更新，因为一个点或者转移到一个新的簇（两次更新），或者留在它的当前簇（零次更新）。使用增量更新策略确保不会产生空簇，因为所有的簇都从单个点开始；并且如果一个簇只有单个点，则该点总是被重新指派到相同的簇。

> 此外，如果使用增量更新，则可以调整点的相对权值；例如，点的权值通常随聚类的进行而减小。尽管这可能产生更好的准确率和更快的收敛性，但是在千变万化的情况下，选择好的相对权值可能是困难的。这些更新问题类以于人工神经网络的权值更新。

接下来就介绍一种最常用，效果也非常好的增量更新质心的方法：二分K-均值法。

四、二分K-均值算法

k-means聚类算法通过不断的更新簇质心直到收敛为止，但是这个收敛是局部收敛到了最小值，并没有考虑全局的最小值。为了克服k-均值算法的局部最小值问题，有人提出了二分K-均值算法。该算法的思想是首先将所有点作为一个簇，然后将该簇一分为二，之后选择其中一个簇继续划分，选择哪一个簇进行划分取决于对其划分是否可以最大程度降低SSE的值。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。

二分k-均值算法的伪代码形式如下：

```
将所有点看做是一个簇
当簇小于数目k时：
    对于每一个簇：
        计算总误差
        在给定的簇上进行2均值聚类
        计算将该簇划分成两个簇后总误差
选择误差最小的那个簇进行划分
重复上述过程直到产生k个簇
```

1. 二分K均值法的python实现

1.1 数据准备

我们仍以前面的test_set为例，数据已经导进来了，这里为了加深印象重新写一遍数据导入代码。

```
#导入相应的包
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#导入数据集
testSet = pd.read_table('testSet.txt', header=None)
testSet.head()
testSet.shape

#给数据集增加一列标签
label = pd.DataFrame(np.zeros(testSet.shape[0]).reshape(-1, 1))
test_set = pd.concat([testSet, label], axis=1, ignore_index = True)
test_set.head()
```

1.2 构建辅助函数

首先是计算整体质心，即对数据集各列求均值

```
n = test_set.shape[1]
centroids = np.array(np.mean(test_set.iloc[:, :n-1]))
```

由于初始质心只有一个，所以总误差平方和可以直接利用distEclud函数计算距离

```
test_SSE = distEclud(test_set.iloc[:, :n-1], centroids).sum()
```

第一次执行切分时只有一个簇（即原始数据集），接下来对原始簇进行二分kMeans聚类，并由聚类结果可计算当前SSE。然后，分别对两个质心进行进一步的有判别的分裂即可。

```
centroids1, result1 = kMeans(test_set, 2)
test_SSE1 = result1.iloc[:, n].sum()
```

考虑到代码可读性，我们首先还是构建一个辅助函数，用于在给定中心点的情况下判断数据集各点划分归属情况，输入参数为数据集、中心点和距离衡量算法。

```
"""
函数功能：在给定质心的情况下划分各点所属簇
参数说明：
    dataSet:原始数据集
    centroids:质心
    distMeas:距离衡量函数
返回：
    result_set：划分结果
"""
def kMeans_assment(dataSet, centroids, distMeas = distEclud):
    m,n= dataSet.shape
    clusterAssment = np.zeros((m,3))
    clusterAssment[:, 0] = np.inf
    clusterAssment[:, 1: 3] = -1
    result_set = pd.concat([dataSet, pd.DataFrame(clusterAssment)], axis=1,
ignore_index = True)
    for i in range(m):
        dist = distMeas(dataSet.iloc[i, :n-1].values, centroids)
        result_set.iloc[i, n] = dist.min()
        result_set.iloc[i, n+1] = np.where(dist == dist.min())[0]
        result_set.iloc[:, -1] = result_set.iloc[:, -2]
    return result_set
```

函数运行结果：

```
result_set = kMeans_assment(test_set, centroids1)
result_set.head()
```

1.3 构建二分K均值函数

接下来编写二分K均值函数

```
def biKmeans(dataSet, k, distMeas = distEclud):
    m,n = dataSet.shape
    centroids, result_set = kMeans(dataSet, 2)
    j = 2
    while j < k:
        result_tmp = result_set.groupby(n+1).sum()
        clusterAssment = pd.concat([pd.DataFrame(centroids), result_tmp.iloc[:,n]],
axis = 1, ignore_index = True)
        lowestSSE = clusterAssment.iloc[:, n-1].sum()
        centList = []
        sseTotle = np.array([])
        for i in clusterAssment.index:
            df_temp = result_set.iloc[:, :n][result_set.iloc[:, -1] == i]
            df_temp.index = range(df_temp.shape[0])
            cent, res = kMeans(df_temp, 2, distMeas)
            centList.append(cent)
            sseSplit = res.iloc[:, n].sum()
            sseNotSplit = result_set.iloc[:, n][result_set.iloc[:, -1] != i].sum()
            sseTotle = np.append(sseTotle, sseSplit + sseNotSplit)
        min_index = np.where(sseTotle == sseTotle.min())[0][0]
        clusterAssment = clusterAssment.drop([min_index])
        centroids = np.vstack([clusterAssment.iloc[:, :n-1].values,
centList[min_index]])
        result_set = kMeans_assment(dataSet, centroids)
        j = j + 1
    return centroids, result_set
```

简单描述上述函数编写思路，首先自定义了一个 `clusterAssment` 的DataFrame作为质心和对应簇的局部误差平方和保存容器，使用DF数据格式主要为了方便使用统计函数和对其进行结构调整，以及保留局部SSE和质心的相互对应关系

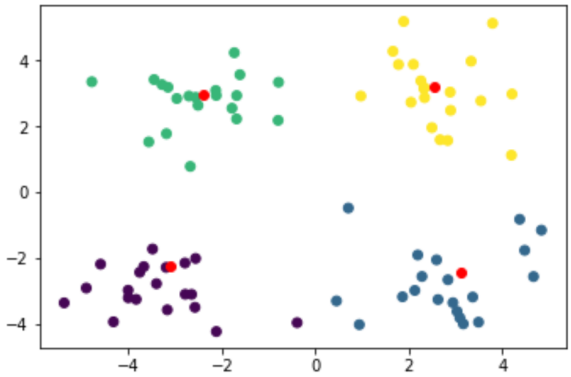
然后从质心为2开始逐渐分割质心，这里以2开始增加质心数量而不是从1开始，主要是为了加快执行效率，毕竟从1开始增加质心要多一次迭代，且毫无意义。外层循环使用while函数控制循环2到k次，每循环一次质心数量增加1，内层循环用for循环，每次循环计算一个质心分裂后总SSE下降的数值，并且使用sseTotle（dnarry容器）保存总SSE，centList（list容器）保存对应质心，并根据最小SSE判断分裂质心。最后使用kMeans_assment方法更新输出结果数据集，投入下一次循环。

2. 模型验证

测试能函数能否正常运行

```
test_cent, test_cluster = biKmeans(test_set, 4)
plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1], c=test_cluster.iloc[:,
-1])
plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red');
```

In [36]:
```
test_cent, test_cluster = biKmeans(test_set, 4)
plt.scatter(test_cluster.iloc[:,0], test_cluster.iloc[:, 1], c=test_cluster.iloc[:, -1])
plt.plot(test_cent[:, 0], test_cent[:, 1], 'o', color='red');
```



五、聚类模型的评价指标

1. 误差平方和SSE

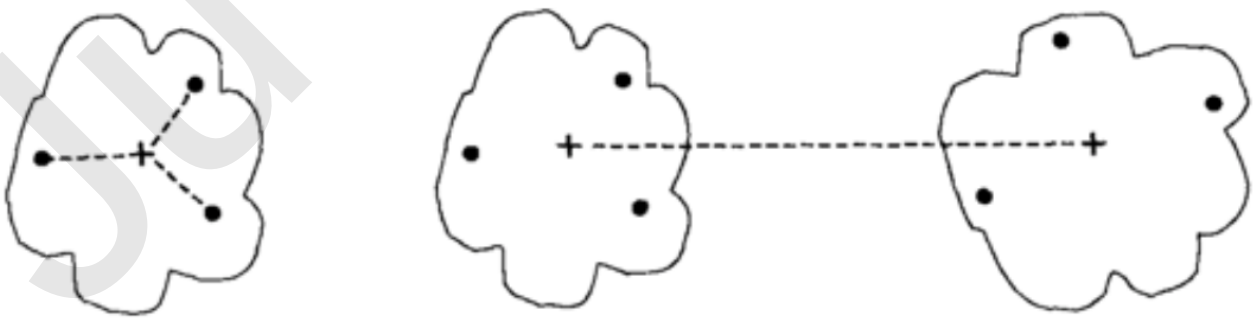
误差平方和（Sum of the Squared Error， **SSE**），也被称为组内误差平方和，它是机器学习中很重要的概念，该概念在聚类和回归类算法中均有广泛应用。在聚类算法中所谓误差平方和是指每个数据点的误差，即它到最所属类别质心的欧几里得距离，然后求和汇总既得误差平方和。在聚类算法中，SSE是我们判断模型是否最优的重要指标，我们希望求得的模型是在给定K值的情况下SSE最小的模型，即在相同的K值情况下聚类模型SSE越小越好，这也是聚类算法最核心的优化条件。

2. 轮廓系数

除了误差平方和SSE，轮廓系数（Silhouette Coefficient），也是聚类效果好坏的一种评价方式。它结合凝聚度和分离度两种因素。下面详细介绍三者关系。

2.1 凝聚度和分离度

对于基于原型的簇，簇的凝聚度可以定义为关于簇原型（质心或中心点）的邻近度的和。同理，两个簇之间的分离度可以用两个簇原型的邻近性度量。如下图所示，其中簇的质心用 “+”标记。



(a) 凝聚度　　(b) 分离度

凝聚度计算公式为

$$cohesion(C_i) = \sum_{x \in C_i} proximity(x, c_i)$$

其中proximity是邻近度计算公式，c_i是簇C_i的质心，当邻近度计算公式取欧几里得距离时，C_i的凝聚度就是该簇的SSE。

分离度的衡量则有两种方法，其一是计算两两质心之间的分离度，可由如下公式计算得出

$$separation(C_i, C_j) = proximity(c_i, c_j)$$

另一种则是计算某簇质心到数据集总体质心之间的分离度

$$separation(C_i) = proximity(c_i, c)$$

其中c是数据集整体质心，进一步我们就可对其进行求和汇总，从而求得总体凝聚度和分离度，这里需要注意，虽然我们在求和过程中可以利用另一套计算体系来对每个分量赋予权值然后求和，但这种做法并不常见，以下讨论均是建立在简单求和汇总的基础上进行的。同时，在简单求和求取总分离度的情况下，上述两种求分离度的方法实际上是等价的。

2.2 凝聚度和分离度的基本性质

- 凝聚度和SSE

  在欧式距离定义的空间内，总凝聚度实际上和误差平方和等价，当然，SSE误差平方和还可被称作组内误差平方和

- 分离度和组间误差平方和（SSR）

  在欧式空间中，当我们采用簇质心和整体质心的邻近度来衡量分离度的时候，实际上总分离度和组间误差平方（SSR）和等价，SSB计算公式如下

$$SSR = \sum_{i=1}^{k} dist(c_i, c)^2$$

- 凝聚度和分离度之间的关系

  实际上，通过数学手段我们能够证明，对于给定的数据集，无论如何划分总SSE和总SSR之和总是一个常数，即离差平方和（SST），这个结果实际上说明要最小化SSE（凝聚度）也就等价于要最大化SSR（分离度），因此之前我们以最小化SSE作为模型优化目标，也可以用凝聚度和分离度的角度进行理解。

  凝聚度和分离度之间的这种相对关系在其他诸多模型中也能遇见，最典型的就是方差分析和回归分析中，我们也常常使用SSE和SSR来衡量模型有效性、提供模型优化的指导意见。

  如在方差分析中，组内SSE就是用以区分连续变量的离散变量各水平所对应的分组内的误差，计算过程和聚类分析也高度类似，其SSE是计算各组内连续变量到其均值之间的距离平方和，而SSR则是各分类水平的均值到整体数据集的均值之间的距离平方和，同时SST则等于数据集中所有数据到均值之间的距离平方和，且SST=SSR+SSE。因此，正确了解SSE和SSR将是我们理解诸多算法的有效途径。

  最后，给出SST=SSR+SSE的证明过程

$$
\begin{aligned}
SST &= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c)^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}\big((x-c_i)-(c-c_i)\big)^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)^2 - 2\sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)(c-c_i)) + \sum_{i=1}^{K}\sum_{x\in C_i}(c-c_i)^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)^2 + \sum_{i=1}^{K}\sum_{x\in C_i}(c-c_i)^2 \\
&= \sum_{i=1}^{K}\sum_{x\in C_i}(x-c_i)^2 + \sum_{i=1}^{K}|C_i|(c-c_i)^2
\end{aligned}
$$

2.3 轮廓系数

轮廓系数（silhouettecoefficient）指标结合了凝聚度和分离度。下面的步骤解释如何计算个体点的轮廓系数。此过程由如下三步组成。我们以欧式距离为例，但是类似的方法可以使用相似度。

（1）对于第*i*个对象，计算它到簇中所有其他对象的平均距离。该值记作a_i。

（2）对于第*i*个对象和不包含该对象的任意簇，计算该对象到给定簇中所有对象的平均距离。关于所有的簇，找出最小值；该值记作b_i。

（3）对于第*i*个对象，轮廓系数是$s_i = \frac{(b_i-a_i)}{max(a_i,b_i)}$

轮廓系数的值在-1和1之间变化。我们不希望出现负值，因为负值表示点到簇内点的平均距离a_i大于点到其他簇的最小平均距离b_i。我们希望轮廓系数是正数（$a_i < b_i$），并且a_i越接近0越好，因为当a_i=0时轮廓系数取其最大值1。我们可以简单地取簇中点的轮廓系数的平均值，计算簇的平均轮廓系数。通过计算所有点的平均轮廓系数，可以得到聚类优良性的总度量。

由此我们能看出，轮廓系数可以在模型取不同质心数量的情况下对模型聚类效果进行纵向比较，进而能够给最终聚类数量提供建议。

3. 轮廓系数的python实现

接下来讨论轮廓系数在Python中的实现方法，首先定义一个用于计算某点到簇内其他点平均距离的函数，由于轮廓系数主要用于聚类算法模型评估中，因此假定给定数据集格式为前文算法导出的结果数据集，其基本格式为

```
centroids, result_set = biKmeans(test_set, 4)
result_set.head()
```

In [37]:
```
centroids, result_set = biKmeans(test_set, 4)
result_set.head()
```

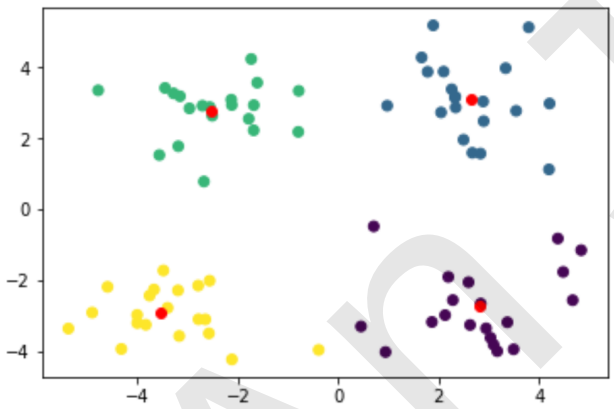
Out[37]:

	0	1	2	3	4	5
0	1.658985	4.285136	0.0	2.320192	1.0	1.0
1	-3.453687	3.424321	0.0	1.261352	2.0	2.0
2	4.838138	-1.151539	0.0	6.638391	0.0	0.0
3	-5.379713	-3.362104	0.0	3.604773	3.0	3.0
4	0.972564	2.924086	0.0	2.769678	1.0	1.0

确认分类效果

```
plt.scatter(result_set.iloc[:,0], result_set.iloc[:, 1], c=result_set.iloc[:, -1])
plt.plot(centroids[:, 0], centroids[:, 1], 'o', color='red');
```

In [38]:
```
plt.scatter(result_set.iloc[:,0], result_set.iloc[:, 1], c=result_set.iloc[:, -1])
plt.plot(centroids[:, 0], centroids[:, 1], 'o', color='red');
```



就该数据集结构，讨论如何计算轮廓系数。其基本思路为，首先在原数据集的基础上增加质心个数的列，每列保留数据为对应点到各簇的点的均值，例如数据集一共有两个质心，则新增两列，数据集中第i个点对应的两列数值分别为点i到簇1和簇2各点的均值，以result_set为例实现该功能

```
result = result_set.copy()
m, n = result.shape
nc = len(centroids)
for i in range(nc):
    result[n+i]=0
result_list = []
for i in range(nc):
    result_temp = result[result.iloc[:, n-1] == i]
    result_temp.index = range(result_temp.shape[0])
    result_list.append(result_temp)
for i in range(m):
    for j in range(nc):
        result.iloc[i,n+j]=distEclud(result.iloc[i, :n-4].values,result_list[j].iloc[:,
:n-4].values).mean()
```

查看执行结果

```
result.head()
```

In [40]:
```
result.head()
```

Out[40]:

	0	1	2	3	4	5	6	7	8	9
0	1.658985	4.285136	0.0	2.320192	1.0	1.0	53.091309	4.136116	20.724456	79.353982
1	-3.453687	3.424321	0.0	1.261352	2.0	2.0	79.588890	38.884591	2.892466	42.233165
2	4.838138	-1.151539	0.0	6.638391	0.0	0.0	9.187702	24.856602	70.302810	72.431709
3	-5.379713	-3.362104	0.0	3.604773	3.0	3.0	69.902615	107.786897	47.834231	5.791472
4	0.972564	2.924086	0.0	2.769678	1.0	1.0	37.885372	4.585602	13.314199	55.069116

可以明显看出，各点到其所属簇的各点平均距离明显小于到其他簇各点的均值。然后计算轮廓系数必要参数a_i和b_i的值

```
result["a"]=0
result["b"]=0
for i in range(m):
    l_temp=[]
    for j in range(nc):
        if(result.iloc[i,n-1] == j):
            result.loc[i,"a"] = result.iloc[i, n+j]
        else:
            l_temp.append(result.iloc[i, n+j])
    result.loc[i,"b"] = np.array(l_temp).min()
```

查看执行结果

```
result.head()
```

In [41]:
```
result["a"]=0
result["b"]=0
for i in range(m):
    l_temp=[]
    for j in range(nc):
        if(result.iloc[i,n-1] == j):
            result.loc[i,"a"] = result.iloc[i, n+j]
        else:
            l_temp.append(result.iloc[i, n+j])
    result.loc[i,"b"] = np.array(l_temp).min()
result["s"] = (result.loc[:,"b"]-result.loc[:,"a"]) /result.loc[:,"a":"b"].max(axis=1)
result.head()
```

Out[41]:

	0	1	2	3	4	5	6	7	8	9	a	b	s
0	1.658985	4.285136	0.0	2.320192	1.0	1.0	53.091309	4.136116	20.724456	79.353982	4.136116	20.724456	0.800423
1	-3.453687	3.424321	0.0	1.261352	2.0	2.0	79.588890	38.884591	2.892466	42.233165	2.892466	38.884591	0.925614
2	4.838138	-1.151539	0.0	6.638391	0.0	0.0	9.187702	24.856602	70.302810	72.431709	9.187702	24.856602	0.630372
3	-5.379713	-3.362104	0.0	3.604773	3.0	3.0	69.902615	107.786897	47.834231	5.791472	5.791472	47.834231	0.878926
4	0.972564	2.924086	0.0	2.769678	1.0	1.0	37.885372	4.585602	13.314199	55.069116	4.585602	13.314199	0.655586

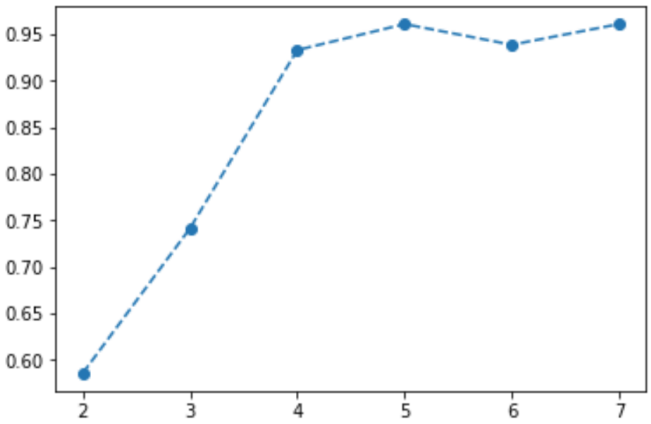
将上述过程写入函数中

```
def silhouetteCoe(result):
    result_set = result.copy()
    m, n = result_set.shape
    nc = len(centroids)
    for i in range(nc):
        result_set[n+i]=0
    result_list = []
    for i in range(nc):
        result_temp = result_set[result_set.iloc[:, n-1] == i]
        result_temp.index = range(result_temp.shape[0])
        result_list.append(result_temp)
    for i in range(m):
        for j in range(nc):
            result_set.iloc[i,n+j]=distEclud(result_set.iloc[i, :n-
4].values,result_list[j].iloc[:, :n-4].values).mean()
    result_set["a"]=0
    result_set["b"]=0
    for i in range(m):
        l_temp=[]
        for j in range(nc):
            if(result_set.iloc[i,n-1] == j):
                result_set.loc[i,"a"] = result_set.iloc[i, n+j]
            else:
                l_temp.append(result_set.iloc[i, n+j])
        result_set.loc[i,"b"] = np.array(l_temp).min()
    result_set["s"] = (result_set.loc[:,"b"]-result_set.loc[:,"a"]) /
result_set.loc[:,"a":"b"].max(axis=1)
    return result_set["s"].mean()
```

进而我们可以利用轮廓系数来在选取不同质心数量时利用K-Means聚类的分类效果，即利用轮廓系数作为聚类K值的学习曲线

```
sil = []
for i in range(1, 7):
    centroids, result_set = biKmeans(test_set, i+1)
    sil.append(silhouetteCoe(result_set))
plt.plot(range(2, 8), sil, '--o')
```

In [47]:
```
sil = []
for i in range(1, 7):
    cent, res = biKmeans(test_set, i+1)
    sil.append(silhouetteCoe(res))
plt.plot(range(2, 8), sil, '--o');
```



同样，从轮廓系数角度而言，4分类或5分类时模型聚类效果最好。当然，轮廓系数同样会收到初始化质心随机性影响，我们可也多次求轮廓系数然后取均值，然后结合SSE学习曲线来共同判别聚类效果最佳的K值。

其他

- 菊安酱的直播间：https://live.bilibili.com/14988341
- 下周一（2019/1/7）将讲解**使用Apriori算法进行关联分析**，欢迎各位进入菊安酱的直播间观看直播
- 如有问题，可以给我留言哦~

【附录1】距离类模型中距离的确定

在距离类模型，例如KNN，KMeans中，有多种常见的距离衡量方法。大多数时候我们使用的是数据距离，但同时，我们在划分法的K-means、基于密度的DBSCAN或者是基于模型的概率方法中也探索文本之间的距离，即文本相似性的度量方式。在机器学习中，一些情况下会定义y表示数据标签，x表示数据特征，而这与众多距离公式中的x，y含义不同，务必避免混淆。

数据距离

- **欧几里得距离**

对于数据之间的距离而言，在欧式空间中我们常常使用欧几里得距离，也就是我们常说的距离平方和开平方。在n维空间中，有两个点x和y，两点的坐标分别为：

$$x\ (x_1, x_2, x_3 \ldots . x_n),\ y\ (y_1, y_2, y_3 \ldots . y_n)$$

则x和y两点之间的欧几里得距离的基本计算公式如下：

$$d(x,y) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \ldots . + (x_n-y_n)^2} = \sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

- **曼哈顿距离**

曼哈顿距离，也被称作街道距离，该距离用以标明两个点在标准坐标系上的绝对轴距总和，其计算方法相当于是欧式距离的1次方表示形式，其基本计算公式如下：

$$d(x,y) = \sum_{i=1}^{n}(|x_i - y_i|)$$

- **闵科夫斯基距离**

无论是欧式距离还是曼哈顿距离，都可视为闵可夫斯基距离的一种特例，该距离计算公式如下：

$$d(x,y) = \sqrt[n]{\sum_{i=1}^{n}(|x_i - y_i|)^n}$$

当n为1时，就是曼哈顿距离，当n为2时即为欧式距离，当n趋于无穷时，即为切比雪夫距离。

文本距离

- **余弦相似度**

而除了数据距离之外，距离的衡量还常见于文本数据之间，此时我们常用余弦相似性或杰卡德相似度来进行衡量。余弦相似度用向量空间中两个向量夹角的余弦值作为衡量两个个体间差异的大小。相比距离度量，余弦相似度更加注重两个向量在方向上的差异，而非距离或长度上。假设两个在二维空间中的向量为x和y，两个向量之间的余弦相似度的基本公式如下:

$$sim(x,y) = cos\theta = \frac{\vec{x} * \vec{y}}{||x|| * ||y||}$$

在n维空间中，余弦相似度的基本计算公式如下

$$cos\theta = \frac{\sum_1^n (x_i * y_i)}{\sqrt{\sum_1^n (x_i)^2} * \sqrt{\sum_1^n (y_i)^2}}$$

当夹角的余弦值越接近于1时，两段文本的距离越近，越相似；夹角的余弦越小，两条文本的距离越远，两者越不相关。

- **杰卡德相似度**

杰卡德相似度则主要以集合运算为主。两个集合A和B交集元素的个数在A、B并集中所占的比例，称为这两个集合的杰卡德系数，用符号 J(A,B) 表示。杰卡德相似系数是衡量两个集合相似度的一种指标（余弦距离也可以用来衡量两个集合的相似度）。

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

与杰卡德相似系数相反的概念是杰卡德距离（Jaccard Distance），可以用如下公式来表示:

$$J_\delta = 1 - J(A,B) = \frac{|A \bigcup B| - |A \bigcap B|}{|A \bigcup B|}$$

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

对于杰卡德相似系数或杰卡德距离来说，它处理的都是非对称二元变量。非对称的意思是指状态的两个输出不是同等重要的，例如，疾病检查的阳性和阴性结果，或银行风控判断中的一个客户是否会违约。

杰卡德相似度算法没有考虑向量中潜在数值的大小，而是简单的处理为0和1，不过，做了这样的处理之后，杰卡德方法的计算效率肯定是比较高的，毕竟只需要做集合操作。

【附录2】归一化方法

对数据集的变量进行归一化处理常见于不同量纲对结果有影响的算法中，而对于基于距离的判别模型而言，但凡涉及到量纲不同，都需要在模型运算之前对数据进行归一化处理，否则量纲大的特征在分类时将占据较大权重，严重时将导致模型失效。

所谓归一化处理，是指将原始各变量的分布通过一定规则整合至相同区间内，从而消除变量的量纲。这里需要注意，虽然偶尔我们会遇到对离散变量消除量纲然后参与距离类判别模型的运算当中，但此处并不规范。离散变量的数值本身并不具备代数意义，最多只具有大小含义（分类变量），对其进行归一化然后进行代数运算并无意义，因此我们在实际分析过程中，应尽量避免该情况的发生。

通常而言，归一化处理主要分为极值标准化（0-1标准化）、Z-score标准化和Sigmoid压缩法，接下来我们对其进行逐一介绍，并利用Python对其进行简单实现

- 0-1标准化

  这是最简单也是最容易想到的方法，同时也是最常用的标准化方法，通过遍历feature vector里的每一个数据，将Max和Min的记录下来，并通过Max-Min作为基数（即Min=0，Max=1）进行数据的归一化处理，基本公式为：

$$x_{normalization} = \frac{x - Min}{Max - Min}$$

  而在Python中，若输入的数据格式为DF，则自定义下列函数对DF的各列进行标准化处理

```
def MaxMinNormalization(dataSet):
    maxDf = dataSet.max()
    minDf = dataSet.min()
    normSet = (dataSet - minDf) / (maxDf - minDf)
    return normSet
```

  其中针对DF的统计函数，max()和min()函数和sum()函数一样，将默认返回针对各列的统计结果，并组成一个Series，若要对行进行统计，则输入参数axis=1即可

```
dfTest = pd.DataFrame({'X1':[1, 2, 3], 'X2':[2, 7, 8]})
dfTest
```

```
In [93]: dfTest = pd.DataFrame({'X1':[1, 2, 3], 'X2':[2, 7, 8]})
         dfTest
```

Out[93]:

	X1	X2
0	1	2
1	2	7
2	3	8

```
dfTest.min()
dfTest.max(1)
```

```
In [95]: dfTest.min()

Out[95]: X1    1
         X2    2
         dtype: int64

In [96]: dfTest.max(1)

Out[96]: 0    2
         1    7
         2    8
         dtype: int64
```

然后利用广播性质，直接对原数据集进行计算即可

```
MaxMinNormalization(dfTest)
```

```
In [97]: MaxMinNormalization(dfTest)

Out[97]:
```

	X1	X2
0	0.0	0.000000
1	0.5	0.833333
2	1.0	1.000000

- Z-score标准化

  和0-1标准化不同，Z-score标准化利用原始数据的均值（mean）和标准差（standard deviation）进行数据的标准化。经过处理的数据将符合标准正态分布，借此完成数据空间压缩，从而消除量纲影响，基本公式为

$$x_{normalization} = \frac{x - \mu}{\sigma}$$

  当输入数据为DataFrame数据格式时Python实现如下

```
def Z_ScoreNormalization(dataSet):
    stdDf = dataSet.std()
    meanDf = dataSet.mean()
    normSet = (dataSet - meanDf) / stdDf
    return normSet
```

  其中，mean()、std()和var()与之前计算的pandas中统计函数一样，当作用于数据框时将直接放回各列的统计结果，mean用于计算均值、std用于计算标准差、var用于计算方差，当要作用于行时需要输入输入axis=1

```
dfTest.var()
dfTest.std()
dfTest.mean()
```

测试自定义数据运行结果

```
Z_ScoreNormalization(dfTest)
```

```
In [100]: Z_ScoreNormalization(dfTest)
```

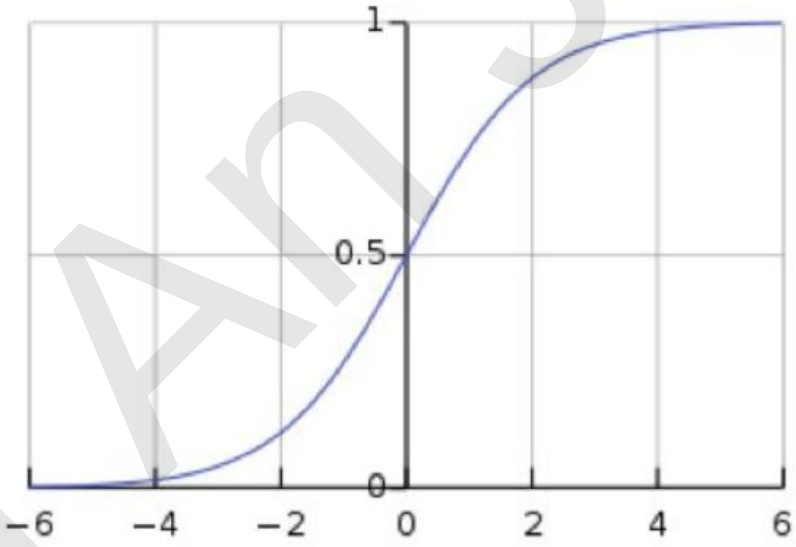
Out[100]:

	X1	X2
0	-1.0	-1.140647
1	0.0	0.414781
2	1.0	0.725866

- Sigmoid压缩法

  Sigmoid函数是一个具有S形曲线的函数，是良好的阈值函数，在(0, 0.5)处中心对称，在(0, 0.5)附近有比较大的斜率，而当数据趋向于正无穷和负无穷的时候，映射出来的值就会无限趋向于1和0，该方法在阈值分割上也有很不错的表现，根据公式的改变，就可以改变分割阈值，该函数是逻辑回归中作为修正函数，同时也是神经网络算法中的激发函数，具体计算公式和函数图像如下所示

$$x_{normalization} = \frac{1}{1+e^{-x}}$$



在Python中实现方法如下所示

```
def sigmodNormalization(dataSet):
    normSet = 1 / (1 + np.exp(-dataSet))
    return normSet
```

测试运行效果

```
sigmodNormalization(dfTest)
```

In [102]:
```
sigmodNormalization(dfTest)
```

Out[102]:

	X1	X2
0	0.731059	0.880797
1	0.880797	0.999089
2	0.952574	0.999665