DBSCAN

【概述】

DBSCAN是Density-Based Spatial Clustering of Applications with Noise的缩写，是一种简单有效的基于密度的聚类算法。

基于密度的聚类旨在检测高密度的区域，这些区域由低密度的区域相互分开

【算法原理】

DBSCAN聚类算法是基于密度的聚类。这种算法假定类别可以通过样本分布的紧密程度来决定。同一类别的样本，它们是比较紧密的，也就是说，对于属于一个类别的样本，在这个样本的不远处很大可能有同一类别的样本。

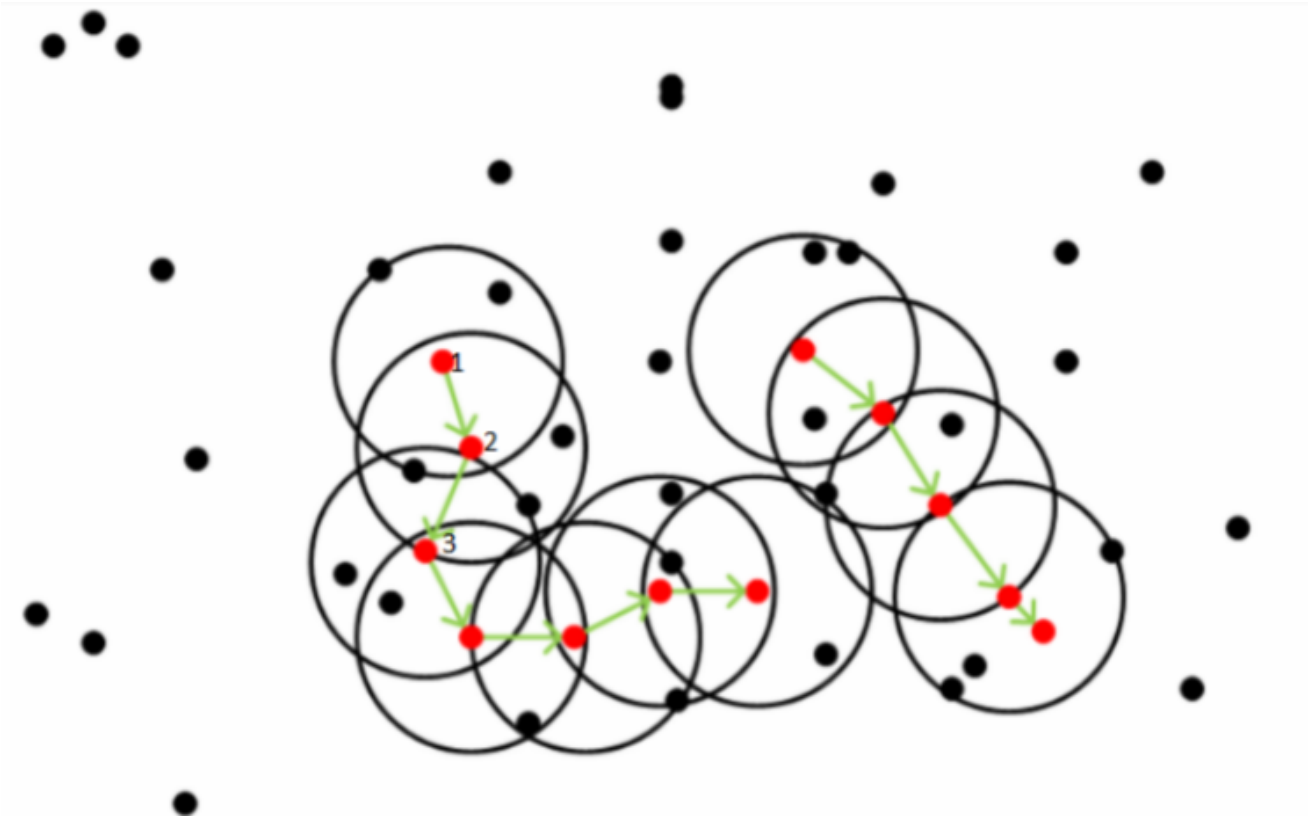
应用DBSCAN算法时，我们需要估计数据集中特定点的密度，特定点的密度是通过计算该点在指定半径下数据点个数（包括特定点），这种计算得到的某个点的密度也被称为局部密度。

计算数据集中每个点的密度时，我们需要把每个点归为以下三类：

1. 如果点的局部密度大于某个阈值，称这个点为核心点。
2. 如果点的局部密度小于某个阈值，但是它落在核心点的邻域内，称这个点为边界点。
3. 如果点不属于核心点页不属于噪声点，称点为噪声点。

除了标记数据集中每个点的类别，我们要做的是根据类别将每个样本进行聚类。对于同一个还未分配的核心点，我们将它邻域内的所有点归为一个新的类C_{new}.如果邻域内有其他核心点的话，我们将重复上面相同情况的动作。

举个例子，下图核心点1邻域内的点归为类C_{new}，因为邻域包括核心点2，那么核心点2邻域内的未分配类别的也归类为C_{new}，核心点2邻域包含核心点3，核心点3也进行与核心点2相同的操作，依次类推（图中绿箭头标识的过程）直到核心点邻域内不包含新的核心点。



【算法流程】

- 如果所有点已经处理，停止
- 对于以前没有处理的特定点，检查它是否是核心点
- 如果不是核心点
  - 将其标记为噪声点
- 如果是核心点，将其标记并
  - 使用这一点形成一个新的聚类C_{new}，并包括集群内的邻域内或边界上的所有点。
  - 将所有这些在邻域内的点插入队列中。
  - 当队列不为空
    - 从队列中删除一个点
    - 如果这个点不是核心点，则将其标记为边界点
    - 如果这个点是核心点，则标记它并检查其邻居中以前没有分配给类的每个点。对于每一未分配的相邻点
      - 将该点分配给当前类C_{new}
      - 将该点插入队列中

【代码实现】

1. 根据上面的算法流程，我们需要求出一个点的邻域内所有的点，目的是判断这点是否为核心点以及处理核心点邻域内的点。

```
def neighbor_points(data, pointId, radius):
    """
    得到邻域内所有样本点的Id
    :param data: 样本点
    :param pointId: 核心点
    :param radius: 半径
    :return: 邻域内所用样本Id
    """
    points = []
    for i in range(len(data)):
        if dist(data[i, 0: 2], data[pointId, 0: 2]) < radius:
            points.append(i)
    return np.asarray(points)
```

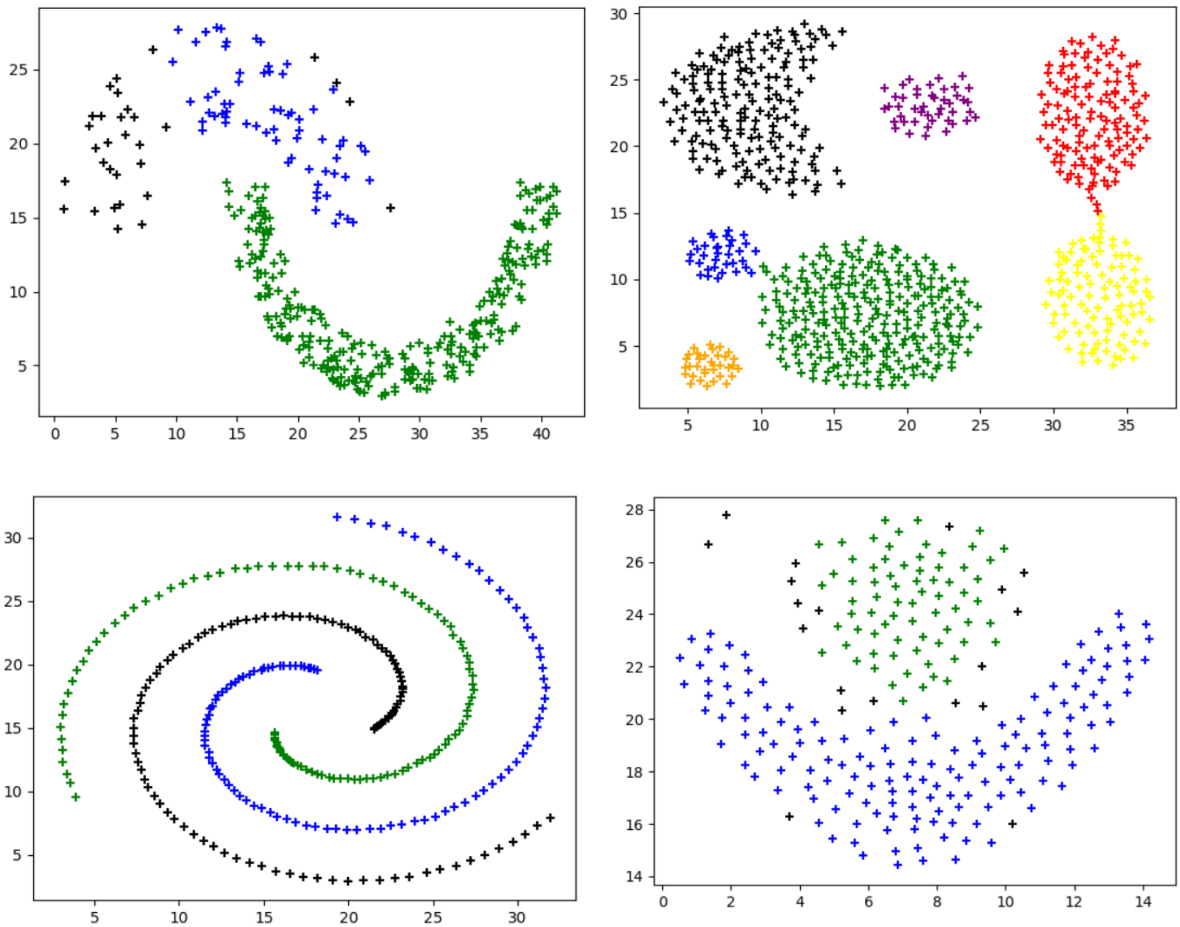
2. 对于一个核心点，我们需要将它和它邻域内所有未分配的样本点分配给一个新类。若邻域内有其他核心点，重复上一个步骤，但只处理邻域内未分配的点，

```
def to_cluster(data, clusterRes, pointId, clusterId, radius, minPts):
    """
    判断一个点是否是核心点，若是则将它和它邻域内的所用未分配的样本点分配给一个新类
    若邻域内有其他核心点，重复上一个步骤，但只处理邻域内未分配的点，并且仍然是上一个步骤的类。
    :param data: 样本集合
    :param clusterRes: 聚类结果
    :param pointId:  样本Id
    :param clusterId: 类Id
    :param radius: 半径
    :param minPts: 最小局部密度
    :return:  返回是否能将点PointId分配给一个类
    """
    points = neighbor_points(data, pointId, radius)
    points = points.tolist()

    q = queue.Queue()

    if len(points) < minPts:
        clusterRes[pointId] = NOISE
        return False
    else:
        clusterRes[pointId] = clusterId
    for point in points:
        q.put(point)
        if clusterRes[point] == UNASSIGNED:
            clusterRes[point] = clusterId

    while not q.empty():
        neighborRes = neighbor_points(data, q.get(), radius)
        if len(neighborRes) > minPts:                       # 核心点
            for i in range(len(neighborRes)):
                resultPoint = neighborRes[i]
                if clusterRes[resultPoint] == UNASSIGNED:
                    q.put(resultPoint)
                    clusterRes[resultPoint] = clusterId
                elif clusterRes[clusterId] == NOISE:
                    clusterRes[resultPoint] = clusterId
    return True
```

3. 扫描整个数据集，为每个数据集打上核心点、边界点和噪声点标签的同时为样本聚类。

```
def dbscan(data, radius, minPts):
    """
    扫描整个数据集，为每个数据集打上核心点，边界点和噪声点标签的同时为
    样本集聚类
    :param data: 样本集
    :param radius: 半径
    :param minPts:  最小局部密度
    :return: 返回聚类结果，  类id集合
    """
    clusterId = 0
    nPoints = len(data)
    clusterRes = [UNASSIGNED] * nPoints
    for pointId in range(nPoints):
        if clusterRes[pointId] == UNASSIGNED:
            if to_cluster(data, clusterRes, pointId, clusterId, radius, minPts):
                clusterId = clusterId + 1
    return np.asarray(clusterRes), clusterId
```

【实验结果】



从上图聚类结果我们可以看出，DBSCAN对空间中任意形状的聚类簇都有比较好的聚类效果。除了有比较好的聚类效果之外，在实验过程中，对比于K-means，DBSCAN不需要确定聚类的个数而且聚类速度快。

图一展示了DBSCAN聚类算法的不足之处：

各个簇之间密度分布不均匀，而且簇之间相距不大时，由于参数半径和局部密度阈值选取困难，导致聚类效果比较差。