机器学习实践作业2 聚类任务

1 任务目标

本次任务目标为使用python编写聚类算法中的K-means、DBSCAN和OPTICS算法。三个算法都使用自定义类实现。

# 2 实现过程

本次实验的实验环境为：

python == 3.8.18

Scipy == 1.10.1

Numpy == 1.20.3

Matplotlib == 3.7.2

## 2.0 数据集的读取与使用

使用numpy库中的loadtxt()函数来读取.dat文件，作为聚类算法的训练数据。因为训练数据本身就是二维的，所以不需要进行降维。

|  |
| --- |
| def create\_dataset(datapath):  #读取dat文件作为聚类数据的数据集  data = np.loadtxt(datapath)  X = data[:, 0:]  #转化X为np.array  X = np.array(X)  return X  datapath = './data/clusterData1.10k.dat'  X\_train = create\_dataset(datapath) |

## 2.1 K-means

### 2.1.1 算法原理

### K-means算法是一种无监督学习算法。它的基本思想是，通过迭代寻找K个簇（Cluster）的一种划分方案，使得每个点到各自的簇中心的距离之和最小。其中，使用欧氏距离来度量每个点到簇中心的距离。

### 2.1.2 实现思路

本次主要使用pytorch中的numpy库来实现K-means算法。初始参数为簇心点个数和迭代次数，首先随机生成指定数量的簇心点，然后在每一次迭代中求出每个样本点距离哪个中心点最近，对各个簇中样本点的坐标加权平均，计算新的中心点，以此来更新簇心的坐标。经过mat\_iter次迭代后，簇心坐标确定，再根据簇心的坐标，来确定每个点分别属于哪个簇，确定每个点的标签。

模型定义

|  |
| --- |
| class KMeans:  def \_\_init\_\_(self,n\_clusters=5,max\_iter=1000):  self.\_n\_clusters=n\_clusters  self.\_X=None  self.\_y=None  self.\_center = None  self.\_max\_iter=max\_iter    def fit(self,X):  self.\_X=X  n=X.shape[0]  d=X.shape[1]  #随机生成中心点  self.\_center = np.array([[np.random.uniform(mi,mx) for mi,mx in zip(X.min(axis=0),X.max(axis=0))] for \_ in range(self.\_n\_clusters)])  step=0  #迭代  while step < self.\_max\_iter:  #求样本点与每个中心点的距离  distances = np.array([np.sum((X-self.\_center[i,:])\*\*2,axis=1) for i in range(self.\_n\_clusters)])  #样本距离哪个最近中心点  self.\_y = np.argmin(distances.T,axis=1)  #对样本点加权平均计算新的中心点  self.\_center = np.array([np.mean(X[self.\_y==i,:],axis=0) for i in range(self.\_n\_clusters)])  step+=1  y\_pred = np.zeros(n)  #确定每个点的标签  for i in range(n):  best\_distance = np.inf  best\_y\_pred = 0  for j in range(self.\_n\_clusters):  distance = np.sum((X[i,:]-self.\_center[j,:])\*\*2)  if distance < best\_distance:  best\_distance = distance  best\_y\_pred = j  y\_pred[i]=best\_y\_pred  return y\_pred |

训练过程

|  |
| --- |
| #使用自己实现的KMeans  n\_clusters = 5  kmeans = KMeans(n\_clusters)  y\_pred = kmeans.fit(X\_train)  y = np.zeros(X\_train.shape[0])  visualize(X\_train, y, y\_pred, title='KMeans', type=None) |

## 2.2 DBSCAN

### 2.2.1 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也不属于边界点，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别，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。

算法流程：

1.如果所有点已经处理，停止

2.对于以前没有处理的特定点，检查它是否是核心点

2.1如果不是核心点

①将其标记为噪声点

2.2如果是核心点，将其标记并

①使用这一点形成一个新的聚类 ，并包括集群内的邻域内或边界上的所有点。

②将所有这些在邻域内的点插入队列中。

③当队列不为空

Ⅰ从队列中取出一个点

Ⅱ如果这个点不是核心点，则将其标记为边界点

Ⅲ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

⑴将该点分配给当前类

⑵将该点插入队列中

2.2.2 实现思路

使用python的numpy，math和queue库，自定义类实现。初始参数为领域半径大小和最小密度，算法流程同上。

|  |
| --- |
| class DBSCAN:  def \_\_init\_\_(self, radius, minPts):  self.data = None  self.radius = radius  self.minPts = minPts  self.NOISE = 0  self.UNASSIGNED = -1  self.clusterId = 1  self.clusterRes = {}    def dist(self,a, b):  return m.sqrt(np.power(a-b, 2).sum())    #获取核心点的邻域内的所有点  def neighbor\_points(self,coreId):  points = []  for i in range(len(self.data)):  if self.dist(self.data[i, 0: 2], self.data[coreId, 0: 2]) < self.radius:  points.append(i)  return np.asarray(points)      def to\_cluster(self,sampleId):  points = self.neighbor\_points(sampleId)  points = points.tolist()  q = queue.Queue()  # 通过密度判断是核心点还是噪声/边界点  if len(points) < self.minPts:  self.clusterRes[sampleId] = self.NOISE  return False  # 若是核心点，将其加入到一个新的类中  else:  self.clusterRes[sampleId] = self.clusterId  # 将核心点邻域内的所有点加入到一个新的类中，并加入遍历队列  for point in points:  if self.clusterRes[point] == self.UNASSIGNED:  q.put(point)  self.clusterRes[point] = self.clusterId  # 对核心点邻域内的点sampleId进行遍历  while not q.empty():  sampleId = q.get()  neighborRes = self.neighbor\_points(sampleId)  #对sampleId邻域内的点进行遍历  if len(neighborRes) >= self.minPts:  for i in range(len(neighborRes)):  resultPoint = neighborRes[i]  # 若该点是核心点，将其邻域内的未分配点加入到新类中,并加入遍历队列  if self.clusterRes[resultPoint] == self.UNASSIGNED:  q.put(resultPoint)  self.clusterRes[resultPoint] = self.clusterId  # 若该点是边界点，直接加入到新类中  elif self.clusterRes[self.clusterId] == self.NOISE:  self.clusterRes[resultPoint] = self.clusterId  return True    def fit(self,data):  self.data = data  nPoints = len(self.data)  #初始化聚类结果  self.clusterRes = [self.UNASSIGNED] \* nPoints  #遍历每一个样本点  for sampleId in range(nPoints):  if self.clusterRes[sampleId] == self.UNASSIGNED:  if self.to\_cluster(sampleId):  self.clusterId = self.clusterId + 1  return np.asarray(self.clusterRes), self.clusterId |

## 2.3 OPTICS

### 2.3.1 算法原理

OPTICS(Ordering points to identify the clustering structure)是一基于密度的聚类算法，OPTICS算法是DBSCAN的改进版本，因此OPTICS算法也是一种基于密度的聚类算法。

OPTICS算法的提出就是为了帮助DBSCAN算法选择合适的参数，降低输入参数的敏感度。OPTICS主要针对输入参数ϵ过敏感做的改进，OPTICS和DBSCAN的输入参数一样（ϵ 和MinPts），虽然OPTICS算法中也需要两个输入参数，但该算法对ϵ输入不敏感（一般将ϵ固定为无穷大），同时该算法中并不显式的生成数据聚类，只是对数据集合中的对象进行排序，得到一个有序的对象列表，通过该有序列表，可以得到一个决策图，通过决策图可以不同ϵ参数的数据集中检测簇集，即：先通过固定的MinPts和无穷大的ϵ得到有序列表，然后得到决策图，通过决策图可以知道当ϵ取特定值时（比如ϵ=3）数据的聚类情况。

OPTICS相对于DBSCAN多了核心距离和可达距离的定义。

核心距离：样本x∈X，对于给定的ϵ和MinPts，使得x成为核心点的最小邻域半径。

可达距离：使得“x成为核心点”和“y可以从x直接密度可达” 同时成立的最小邻域半径。

算法流程：

1.初始化核心点集合Ω=∅

2.遍历样本集X的元素，如果是核心点，则将其加入到核心点集合Ω中

3.如果核心点集合Ω中元素都已经被处理，则算法结束，否则转入步骤4.

4.在核心点集合Ω中，随机选择一个未处理的核心点o，首先将o标记为已处理，同时将o压入到有序列表p中，最后将o的ϵ-邻域中未访问的点，根据可达距离的大小（计算未访问的邻居点到o点的可达距离）依次存放到种子集合seeds中。

5.如果种子集合seeds=∅，跳转到3，否则，从种子集合seeds中挑选可达距离最近的种子点seed，首先将其标记为已访问，首先将seed标记为已处理，同时将seed压入到有序列表p中，然后判断seed是否为核心对象，如果是将seed中未访问的邻居点加入到种子集合中，重新计算可达距离。（计算种子集合中距离seed点的可达距离）跳转到5。

6.将计算的可达距离按照有序列表进行排序，找到有序列表中可达距离小于eps的点的索引，依此索引进行分类。

### 2.3.2 实现思路

使用python中的numpy，scipy库，自定义OPTICS类实现。算法流程同上。

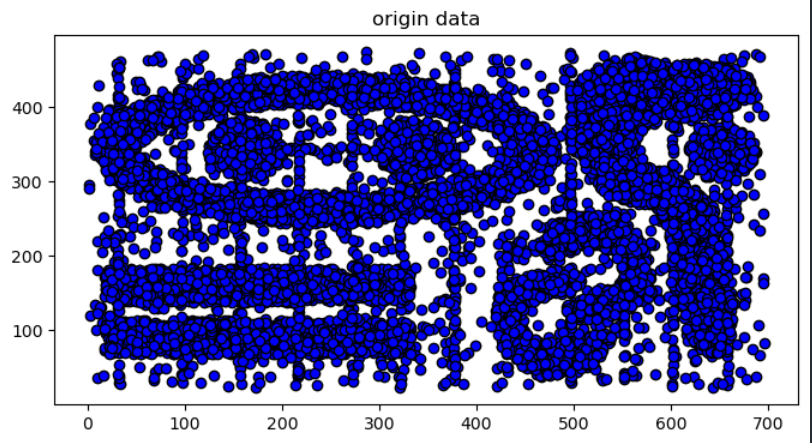
|  |
| --- |
| class OPTICS:  def \_\_init\_\_(self,eps=np.inf,minPts=15):  self.data = None  self.eps = eps  self.minPts = minPts  self.seeds = {}  self.isProcess = None  self.disMat = None  self.reach\_dists = None  self.core\_dists = None  # 计算距离矩阵  def distM(self):  return squareform(pdist(self.data,metric='euclidean'))    def updateSeeds(self,core\_PointId,neighbours):  disMat = self.disMat  isProcess = self.isProcess  core\_dists = self.core\_dists  # 获得核心点core\_PointId的核心距离  core\_dist=core\_dists[core\_PointId]  # 遍历core\_PointId 的每一个邻居点  for neighbour in neighbours:  # 如果neighbour没有被处理过，计算该核心距离  if(isProcess[neighbour]==-1):  # 首先计算该点针对core\_PointId的可达距离  new\_reach\_dist = max(core\_dist, disMat[core\_PointId][neighbour])  # 如果该点的可达距离未定义，即为未访问过，将其放入seeds中  if(np.isnan(self.reach\_dists[neighbour])):  self.reach\_dists[neighbour]=new\_reach\_dist  self.seeds[neighbour] = new\_reach\_dist  # 如果该点的可达距离已经定义，比较新计算的可达距离和原有的可达距离，将小的放入seeds  elif(new\_reach\_dist<self.reach\_dists[neighbour]):  self.reach\_dists[neighbour] = new\_reach\_dist  self.seeds[neighbour] = new\_reach\_dist  return self.seeds    #计算数据集中的可达列表，及其可达距离  def compute\_optics(self):  eps = self.eps  minPts = self.minPts  data = self.data  # 获得距离矩阵  self.disMat = self.distM()  disMat = self.disMat  #print(f'disMat.shape:{disMat.shape}')  orders = []  # 获得数据的行和列(一共有n条数据)  n, m = data.shape  # 初始化可达距离，处理标记  self.reach\_dists= np.full((n,), np.nan)  self.isProcess = np.full((n,), -1)  # 找到每个点距离矩阵中距离从小到大排序后第minPTS个元素的索引，然后按照索引计算每个点到第minPTS个元素的距离，为核心距离  minPts\_index = np.argsort(disMat)[:,minPts-1]  temp\_core\_distances = disMat[np.arange(0,n),minPts\_index]  # 核心距离，小于eps的为核心点，大于eps的为非核心点  self.core\_dists = np.where(temp\_core\_distances <= eps, temp\_core\_distances, -1)      # 将disMat矩阵的中小于eps的数赋予1，大于eps的数赋予零(list1)，对每一行求和,数量大于minPTs的就是核心点  list1 = np.where(disMat <= eps, 1, 0)  core\_points\_index = np.where(np.sum(list1, axis=1) >= minPts)[0]  print(f'core\_points\_index:{core\_points\_index}')    # 遍历所有的核心点  for pointId in core\_points\_index:  # 如果核心点未被分类，将其作为的种子点，开始寻找相应簇集  if (self.isProcess[pointId] == -1):  # 将点pointId标记为当前类别(即标识为已操作)  self.isProcess[pointId] = 1  orders.append(pointId)  # 寻找种子点的eps邻域且没有被分类的点，将其放入种子集合  neighbours = np.where((disMat[:, pointId] <= eps) & (self.isProcess == -1))[0]  seeds = dict()  # 用该核心点和它的领域更新种子点  seeds= self.updateSeeds(pointId,neighbours)  # 只要种子集合非空，就一直循环  while len(seeds)>0:  # 从种子集合中取出第一个点  next\_pointId = sorted(seeds.items(), key=operator.itemgetter(1))[0][0]  del seeds[next\_pointId]  self.isProcess[next\_pointId] = 1  orders.append(next\_pointId)  # 寻找next\_point 种子点eps邻域（包含自己）  next\_neighbours = np.where(disMat[:, next\_pointId] <= eps)[0]  if len(next\_neighbours) >= minPts:  seeds=self.updateSeeds(next\_pointId,next\_neighbours)  # 簇集生长完毕，寻找到一个类别  # 返回数据集中的可达列表，及其可达距离  return orders,self.reach\_dists    def fit(self,data):  self.data = data  orders,reach\_dists = self.compute\_optics()  print(f'len(orders):{len(orders)}')  print(orders)  print(f'len(reach\_dists):{len(reach\_dists)}')  print(reach\_dists)  eps = self.eps  # 获得数据的行和列  n,m=data.shape  # reach\_dists[orders] 将每个点的可达距离，按照有序列表排序（即输出顺序）  # np.where(reach\_dists[orders] <= eps)[0]，找到有序列表中可达距离小于eps的点的索引，即对应有序列表的索引  reach\_distIds=np.where(reach\_dists[orders] <= eps)[0]  print(f'len(reach\_distIds):{len(reach\_distIds)}')  print(reach\_distIds)  # current为当前点，pre为前一个点，clusterId为类别  pre=reach\_distIds[0]-1  clusterId=0  labels=np.full((n,),-1)  for current in reach\_distIds:  # 如果current-pre!=1，说明是一个新的类别  if(current-pre!=1):  # 类别+1  clusterId=clusterId+1  labels[orders[current]]=clusterId  pre=current  return labels |

# 3 训练结果及对比分析

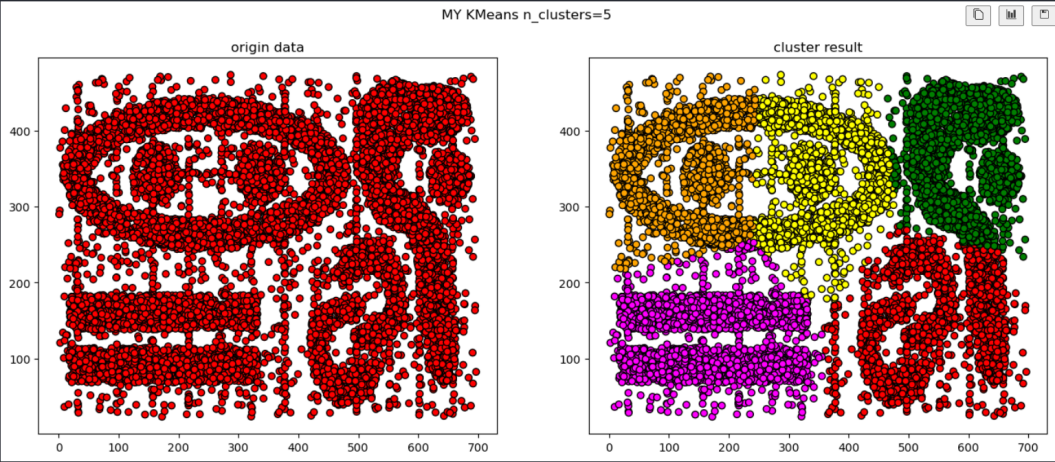
## 3.1 K-means

自定义类的K-means算法可以达到和sklearn库里的K-means算法同样的聚类结果。

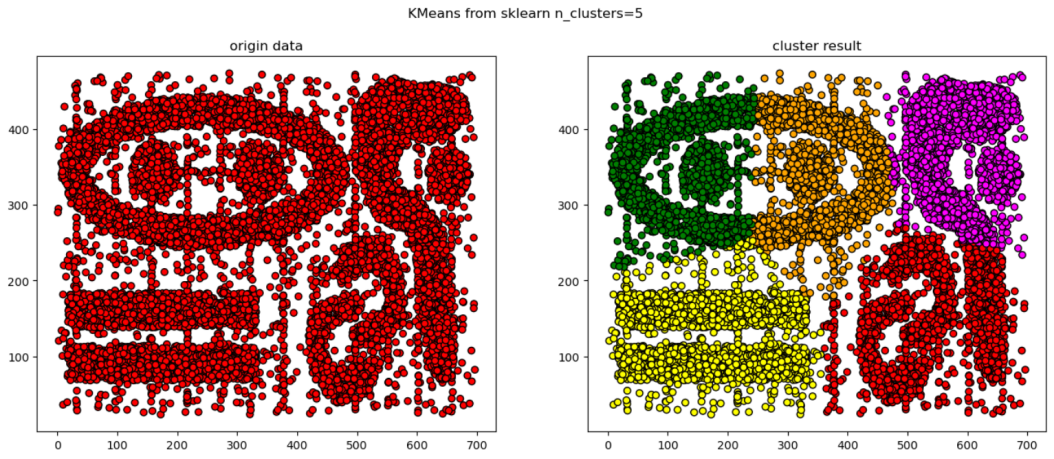
对下列数据集（含10k个数据点），自定义K-means算法的执行时间是2.0s，sklearn库的K-means算法的执行时间是0.6s。可以看到自定义的K-means算法的并行性没有sklearn库好。



**Fig. 1.**原始数据预览



**Fig. 2.自定义K-means聚类结果**



**Fig. 3.sklearn K-means聚类结果**

可以发现，K-means得到的簇更偏向于球形，由于K-means算法是基于层次划分的聚类算法，对于这种非凸集合的数据，不能够很好地识别相应的簇。

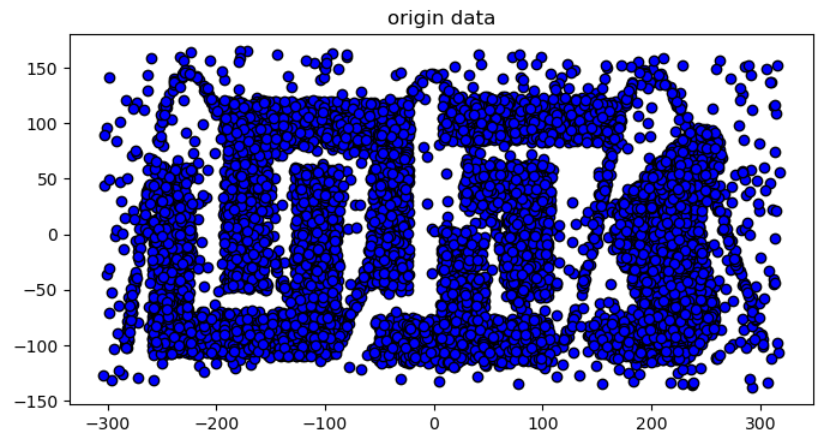
另外，K-means算法需要预先确定聚类的个数。

并且，由于K-means算法对初始选取的聚类中心点敏感，而自定义算法中随机选取中心点，所以可能会造成聚类个数相同而聚类结果不同的情况。

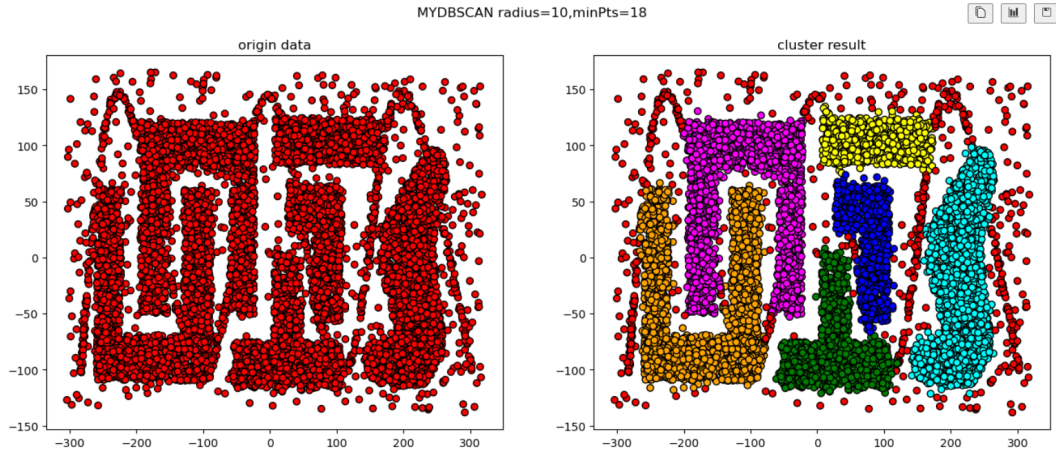
## 3.2 DBSCAN

自定义类的DBSCAN算法可以达到和sklearn库里的DBSCAN算法同样的聚类结果，但执行时间较长。

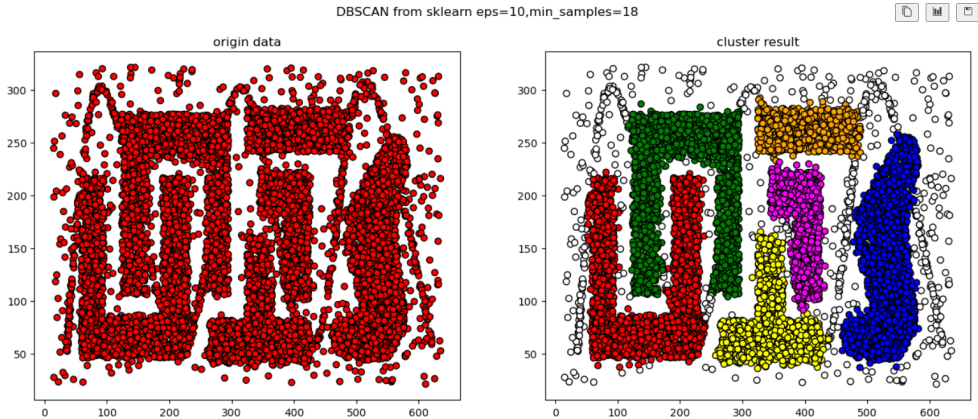
对下面的数据集，自定义DBSCAN算法的执行时间是4min，sklearn库的DBSCAN算法的执行时间是0.5s。这可能是因为，自定义DBSCAN算法中使用了大量遍历操作，同一时间只进行了一次遍历，而sklearn库中可以并行地遍历多个元素。



**Fig. 4.原始数据预览**



**Fig. 5.自定义DBSCAN聚类结果**



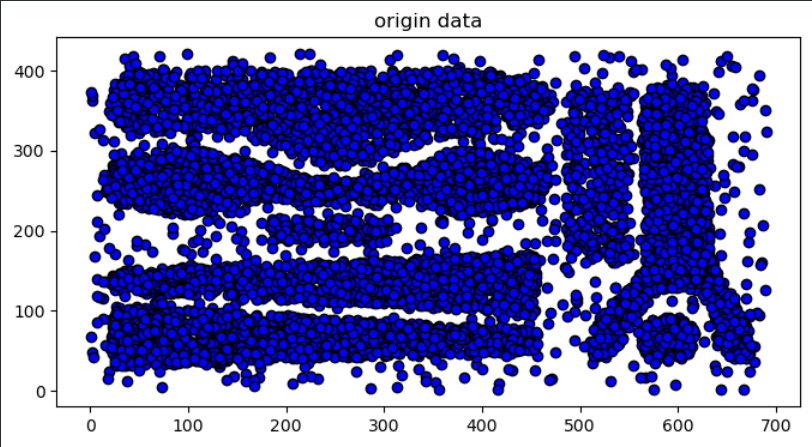
**Fig. 6.sklearn DBSCAN聚类结果**

由于DBSCAN算法对于参数非常敏感，需要花费大量时间调整参数才能达到理想的效果，而参数的设置往往依据经验，所以该算法具有一定的局限性。但可以看到该算法对于任意形状的聚类簇都有较好的聚类效果。

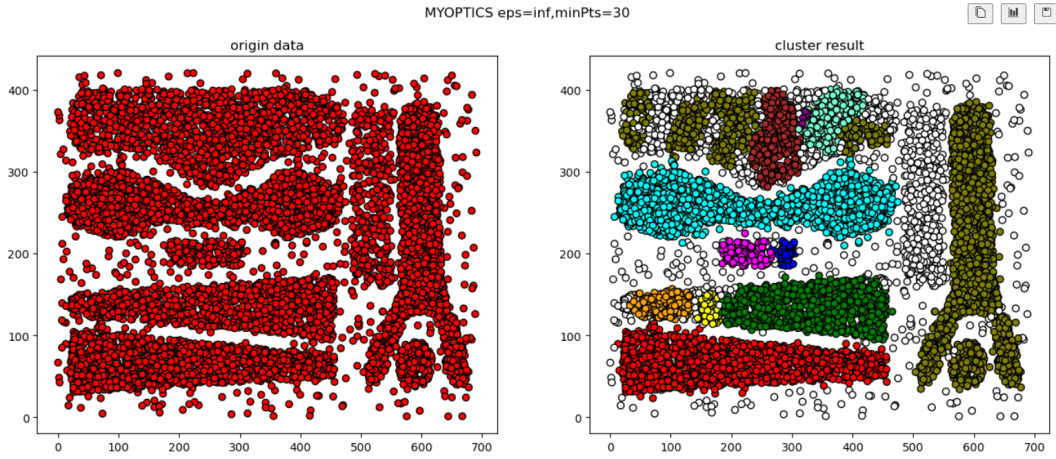
## 3.3 OPTICS

自定义类的OPTICS算法效果比sklearn库里的OPTICS算法聚类效果更好，但执行时间较长。并且，无论如何调整参数，两个算法都不能够很好地对所有聚类簇合理的聚类。

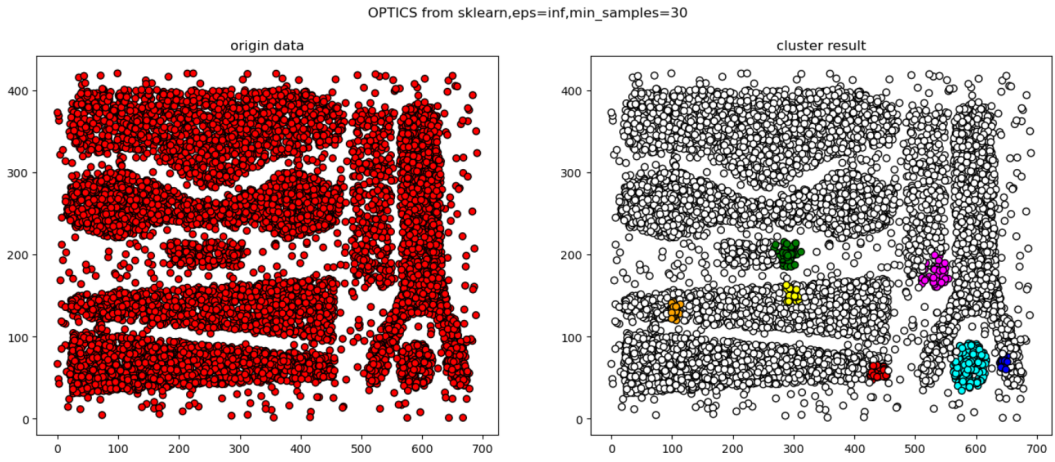
对下面的数据集，自定义OPTICS算法的执行时间是1min21s，sklearn库的DBSCAN算法的执行时间是4.9s。这可能是因为，自定义OPTICS算法中使用了大量遍历操作，同一时间只进行了一次遍历，而sklearn库中可以并行地遍历多个元素。



**Fig. 7**原始数据预览



**Fig.8.**自定义OPTICS算法聚类结果



**Fig.9.**sklearn OPTICS算法聚类结果

# 4 总结

聚类算法的种类非常繁多，有层次聚类方法、划分聚类方法、基于密度的聚类方法、基于子空间的聚类方法、人工神经网络聚类方法等。本次实践通过自行实现并对比K-means(层次聚类)、DBSCAN(基于密度的聚类)和OPTICS(基于密度的聚类),来探索不同聚类算法各自的优缺点。

本次实验最大的感受就是，对于这些比较基础的聚类算法，参数的选取对于聚类结果的影响是非常大的，有时候minPts只毫厘之差，聚类效果却失之千里。而这些参数又往往基于人对于数据的观察和经验，从某种方面来说，这也是人给出的一种”标签”。OPTICS算法本来想要减小EPS对于聚类效果的影响，但可能算法本身具有局限，聚类效果反而没有DBSCAN算法好。

但这并不代表OPTICS算法本身差于DBSCAN，或者K-means算法本身差于DBSCAN，因为不同的算法对应的适宜数据集往往是不同的，而本次所使用的数据集均为由非凸集合构成的数据集。实际应用中，对于不同的数据集，选择合适的聚类算法，方为上策。

References

1. <https://en.wikipedia.org/wiki/K-means_clustering>
2. <https://zh.wikipedia.org/zh-tw/DBSCAN>
3. [KMeans聚类算法详解 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/184686598)
4. [【数据挖掘】基于密度的聚类方法 - OPTICS 方法 ( 算法流程 | 算法示例 )\_optics算法流程-CSDN博客](https://blog.csdn.net/shulianghan/article/details/105938236?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-105938236-blog-133363454.235^v43^pc_blog_bottom_relevance_base8&spm=1001.2101.3001.4242.1&utm_relevant_index=3)
5. [OPTICS算法原理解析与实现\_optics 算法-CSDN博客](https://blog.csdn.net/universsky2015/article/details/133363454?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-133363454-blog-85049135.235^v43^pc_blog_bottom_relevance_base8&spm=1001.2101.3001.4242.1&utm_relevant_index=1)