聚类作业

小组成员：

莫易川
郦洋
黎锦灏
李铁

一、数据源选取

数据集名称：**MNIST手写数字数据集**
获取地址：http://yann.lecun.com/exdb/mnist/

二、数据分析

我们选用 **MNIST数据集** 的训练集部分作为本次分析的对象。MNIST 数据集来自美国国家标准与技术研究所, National Institute of Standards and Technology (NIST). 训练集 (training set) 由来自 250 个不同人手写的数字构成, 其中 50% 是高中学生, 50% 来自人口普查局 (the Census Bureau) 的工作人员.测试集(test set) 也是同样比例的手写数字数据.

在 MNIST 数据集中的每张图片由 28 x 28 个像素点构成, 每个像素点用一个灰度值表示. 若将向量打平，则每张图由784个像素值表示。数据一共分为 10 个类，分别为数字 0 到数字 9。MNIST 的训练集包含 60,000 张手写数字图，大小共 47MB ，其中 10 个类各有 6000 个样本。我们使用了 MNIST 训练集的前 10,000 张图片作为样本。

MNIST 数据集相对于简单数据向量而言具有更大的复杂性，但其分类十分明确且数据的分布规律相对简单，可视为在高维空间中的低维流形。这种特征使得 MNIST 数据集降维到低维空间可以仍然保持分布特性，可以得到较好的降维效果以供聚类等方法做进一步处理。为了得到较好的可视化效果，我们希望 我们用以聚类的特征是相对低维的，可以视觉上明确感知的，MNIST 恰好符合这一特点。我们能够通过降维方法来获得 MNIST 在低维空间中的投影，再对其进行聚类，得到相应数据的分类。

同时，MNIST 数据集还具有标签信息，因此我们可以利用标签信息对聚类结果进行验证，更好评估聚类的效果。

使用 TensorFlow 中 `input_data.py` 脚本来读取数据及标签，给出读取代码如下：

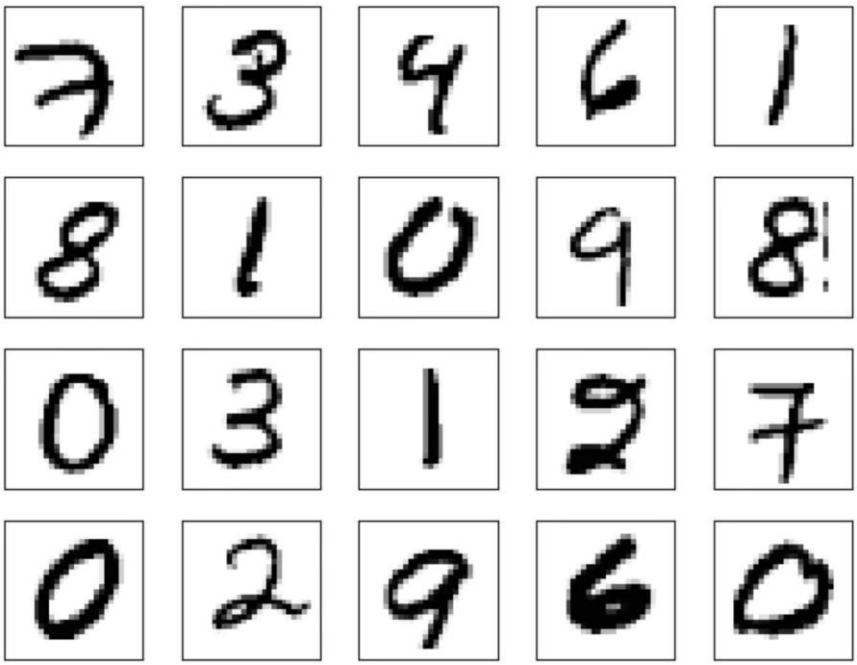
```
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt

mnist = input_data.read_data_sets('MNIST_data',one_hot=True)      # MNIST_data指的是存放数据的文件夹路径，one_hot=True 为采用one_hot的编码方式编码标签

#load data
train_X = mnist.train.images                #训练集样本
validation_X = mnist.validation.images      #验证集样本
test_X = mnist.test.images                  #测试集样本
#labels
train_Y = mnist.train.labels                #训练集标签
validation_Y = mnist.validation.labels      #验证集标签
```

```
test_Y = mnist.test.labels                          #测试集标签

print(train_X.shape,train_Y.shape)                  #输出训练集样本和标签的大小

#可视化样本，下面是输出了训练集中前20个样本
fig, ax = plt.subplots(nrows=4,ncols=5,sharex='all',sharey='all')
ax = ax.flatten()
for i in range(20):
    img = train_X[i].reshape(28, 28)
    ax[i].imshow(img,cmap='Greys')
ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()
```

输出结果如下：

```
(55000, 784) (55000, 10)          #训练集样本和标签的大小
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]    #第一个样本的标签，one-hot编码
```

训练集前20个样本：



三、数据预处理

对于读取的 MNIST 图片，我们没有做进一步处理，直接读取为 784 维向量供后续处理。为了适应聚类的任务需求和进行更好的可视化，我们希望通过降维的方法将 MNIST 数据集中单图 784 维向量投影至二维空间，这里我们选用 **t-SNE 算法** 进行 **特征降维**。

t-SNE算法的基本思想为希望源空间数据分布和投影空间的数据分布趋同，通过梯度下降方法来优化投影空间的数据分布与源分布的一致性。

算法中源空间数据的概率分布通过计算各点间相似度占所有相似度之和比重得到：

$$P(x^j|x^i) = \frac{S(x^i, x^j)}{\sum_{k \neq 1} S(x^i, x^k)}$$

其中，$S(x^i, x^j)$表示源空间中两个向量的相似度。

投影空间中数据的概率分布通过相似的方式计算得到：

$$Q(z^j|z^i) = \frac{S'(z^i, z^j)}{\sum_{k \neq 1} S'(z^i, z^k)}$$

其中，$S'(x^i, x^j)$表示源空间中两个向量的相似度。

在优化时，算法将优化 z使得两个分布之间的距离尽可能近，其中两个分布的距离度量采用KL散度：

$$L = \sum_i KL(P(*|x^i)||Q(*|z^i)) = \sum_i \sum_j P(x^j, x^i) \log \frac{P(x^j|x^i)}{Q(z^j|z^i)}$$

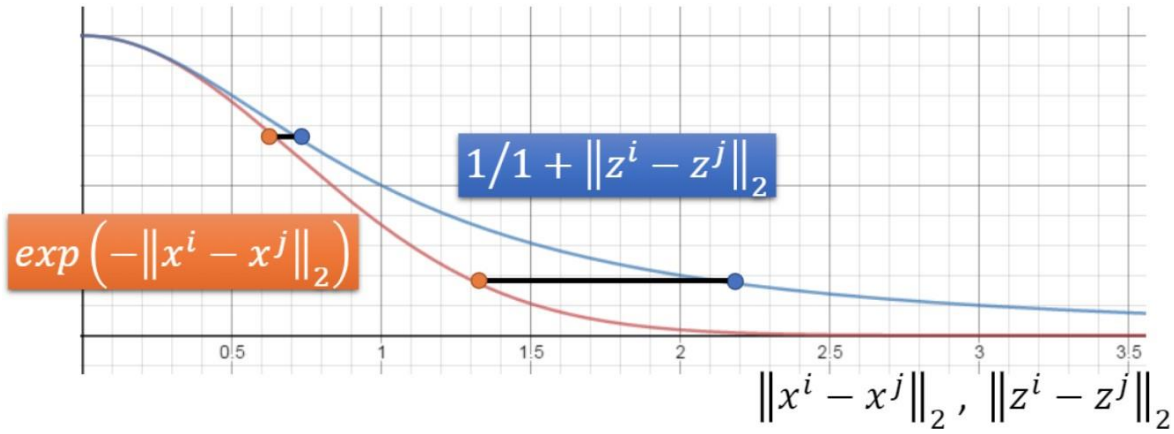
&e msp; 在t-SNE 算法中，特别对相似度函数做了设计，在源空间中，相似度函数定义为：

$$S(x^i, x^j) = \exp(-||x^i - x^j||_2)$$

而在投影空间中，相似度函数定义为：

$$S'(z^i, z^j) = \frac{1}{1 + ||z^i - z^j||_2}$$

在相同相似度下，由于 S 函数相较于S'函数对于距离的敏感性更强，若要保证两个空间中分布的一致性，能够保证在投影空间中两个向量之间的距离较源空间中更大，从而形成的降维簇间具有天然的分隔距离，更加适应聚类任务。



四、聚类处理

4.1 K-Means

K-Means 聚类算法（K-Means Clustering Algorithm）是一种 **基于分割的迭代求解聚类分析算法。**其主要的特点有：

- 原理比较简单，实现比较容易，收敛速度快
- 需要预先确定聚类的个数
- 对初始选取的聚类中心点敏感，得到的结果一般为局部的最优解
- 对噪声数据和异常值敏感
- 处理非球形簇的聚类问题时表现较差

该算法实际上是寻找合适中心点并将样本分配到k个中心点的过程。算法首先初始化k个中心点并将样本分配到离它最近的中心点，然后在新的聚类中重新计算各个聚类的中心点，这样不断更新k个中心点直到收敛。

算法步骤:

首先随机选取k个点作为初始中心点　　a)
将每个点指派到最近的质心，形成k个簇b)
重新计算每个簇的中心点
重复a、b直至簇不发生变化或达到预设的迭代次数

代码实现:

计算两个矩阵的距离矩阵：

```
def compute_distances_no_loops(A, B):
    return  cdist(A,B,metric='euclidean')
```

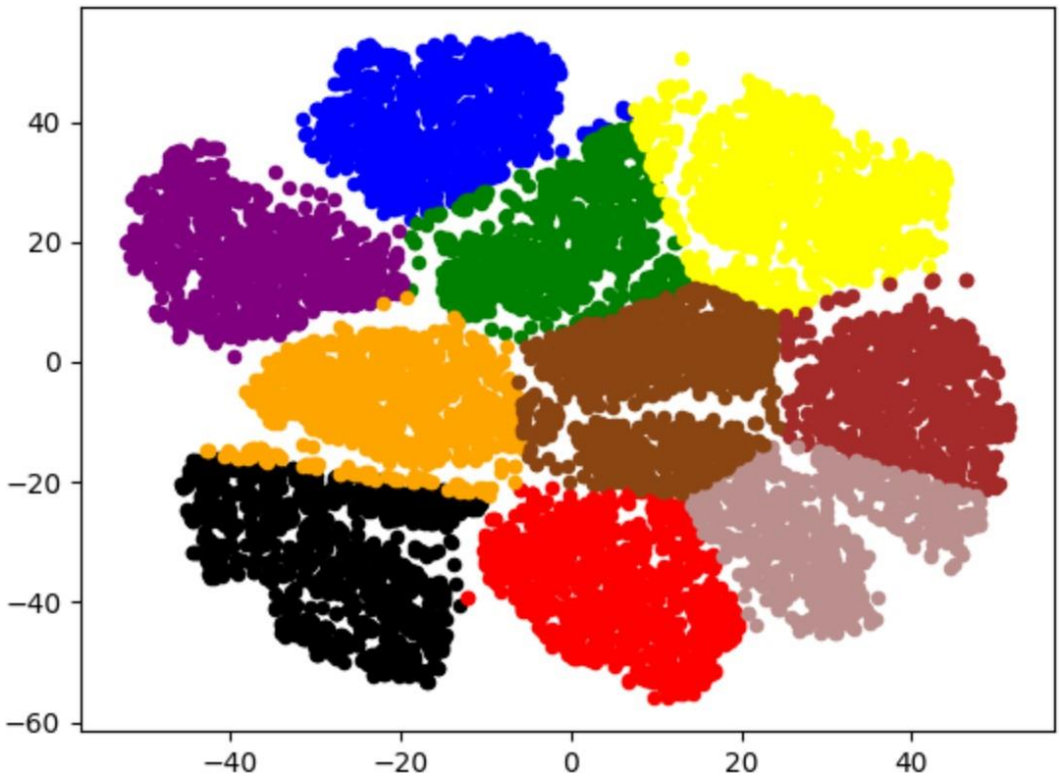
K-Means算法核心过程：

```
def  Kmeans(data,K,tol,N):
    #获取数据的数量
    n = np.shape(data)[0]
    # 从n条数据中随机选择K条，作为初始中心点
    centerId = random.sample(range(0, n), K)
    centerPoints = data[centerId]
    #  计算data到centerPoints的距离矩阵
    dist = compute_distances_no_loops(data, centerPoints)
    # axis=1寻找每一行中最小值都索引
    #  getA()是将mat转为ndarray
    #  squeeze()是将label压缩成一个列表
    labels = np.argmin(dist, axis=1).squeeze()
    # 初始化old J
    oldVar = -0.0001
    # 计算new J
    newVar = np.sum(np.sqrt(np.sum(np.power(data - centerPoints[labels], 2),
axis=1)))
    # 初始化迭代次数
    count=0
    #  当ΔJ大于容差且循环次数小于迭代次数，一直迭代。否则结束聚类
    while count<N and abs(newVar - oldVar) > tol:
        oldVar = newVar
        for i in range(K):
            #  重新计算每一个类别都中心向量
            centerPoints[i] = np.mean(data[np.where(labels == i)], 0)
        # 重新计算距离矩阵
        dist = compute_distances_no_loops(data, centerPoints)
        # 重新分类
        labels = np.argmin(dist, axis=1).squeeze()
        # 重新计算J
        newVar = np.sum(np.sqrt(np.sum(np.power(data - centerPoints[labels], 2),
axis=1)))
        # 迭代次数加1
```

```
            count+=1
        # 返回类别标识，中心坐标
        return  labels,centerPoints
```

将聚类后的数据可视化显示：

```
def plotFeature(data, labels_):
    clusterNum=len(set(labels_))
    fig = plt.figure()
    scatterColors = ['black', 'blue', 'green', 'yellow', 'red', 'purple',
'orange',  'brown','#BC8F8F','#8B4513','#FFF5EE']
    ax = fig.add_subplot(111)
    for i in range(-1,clusterNum):
        colorSytle = scatterColors[i % len(scatterColors)]
        subCluster = data[np.where(labels_==i)]
        ax.scatter(subCluster[:,0], subCluster[:,1], c=colorSytle, s=20)
    plt.show()
```



可以看到该图中聚类结果虽然有部分点存在明显偏离中心点，但是在总体上基本和预期结果差距不大。由于 K-Means 算法对初始中心点的选取比较敏感，上图是经过几次实验选取的效果较好的聚类结果，但总得来说，聚类的结果并没有较大的差别。

4.2 DBSCAN

DBSCAN（Density-Based Spatial Clustering of Applications with Noise）是一种**基于密度**且具有噪声的聚类方法，其主要特点有

- 聚类初始无需指定簇的个数
- 对数据库样本顺序不敏感
- 可以发现任意形状的簇类
- 能够识别出噪声点

DBSCAN 算法定义密度为**聚类空间的一定区域内所包含对象数目**，将密度大于阈值的点所在区域划分为一**簇**，并在具有噪声的空间数据库中发现任意形状的簇。在DBSCAN算法中，簇可以被定义为**密度相连的点的最大集合。**

在 DBSCAN 算法中，我们重点考察两个参数：**Eps 和 MinPts**：

- **Eps**：邻域扫描半径
- **MinPts**：作为核心点的邻域最小对象数目（即以核心点为圆心，Eps为半径的圆内点数）

在 DBSCAN 算法中，数据点被分为三类：

- **核心点**：在邻域内含有超过 MinPts 数目对象的点。
- **边界点**：邻域内对象数目小于 MinPts ，但是落在核心点邻域内的点。
- **噪音点**：既不属于核心点，也不属于边界点。

算法步骤:

- 首先遍历所有点，寻找邻域内含有超过 MinPts 数目对象的点，定义为核心点。
- 对于每个核心点，拓展其邻域内点，标记 LabeIs 为当前簇。
- 若新拓展点为核心点，则继续寻找与新拓展点距离小于 Eps 的点。
- 重复上述操作并扩展为当前聚类簇。

算法优点

1. DBSCAN 是基于密度的聚类算法，能识别出噪声点。
2. DBSCAN 可以发现任意形状的簇类。
3. DBSCAN 无需预先指定要形成的簇类数目。
4. DBSCAN 对于数据库样本顺序不敏感。

算法缺点

1. 若密度分布不均匀或聚类间距大，则聚类效果较差。
2. DBSCAN 不能很好反映高维数据集。
3. 对 I/O 、内存的要求随数据量增长快。
4. DBSCAN 对输入参数非常敏感

代码分析:

计算距离矩阵:

```
# 计算距离矩阵
def  compute_squared_EDM(X):
  return  squareform(pdist(X,metric='euclidean'))
```

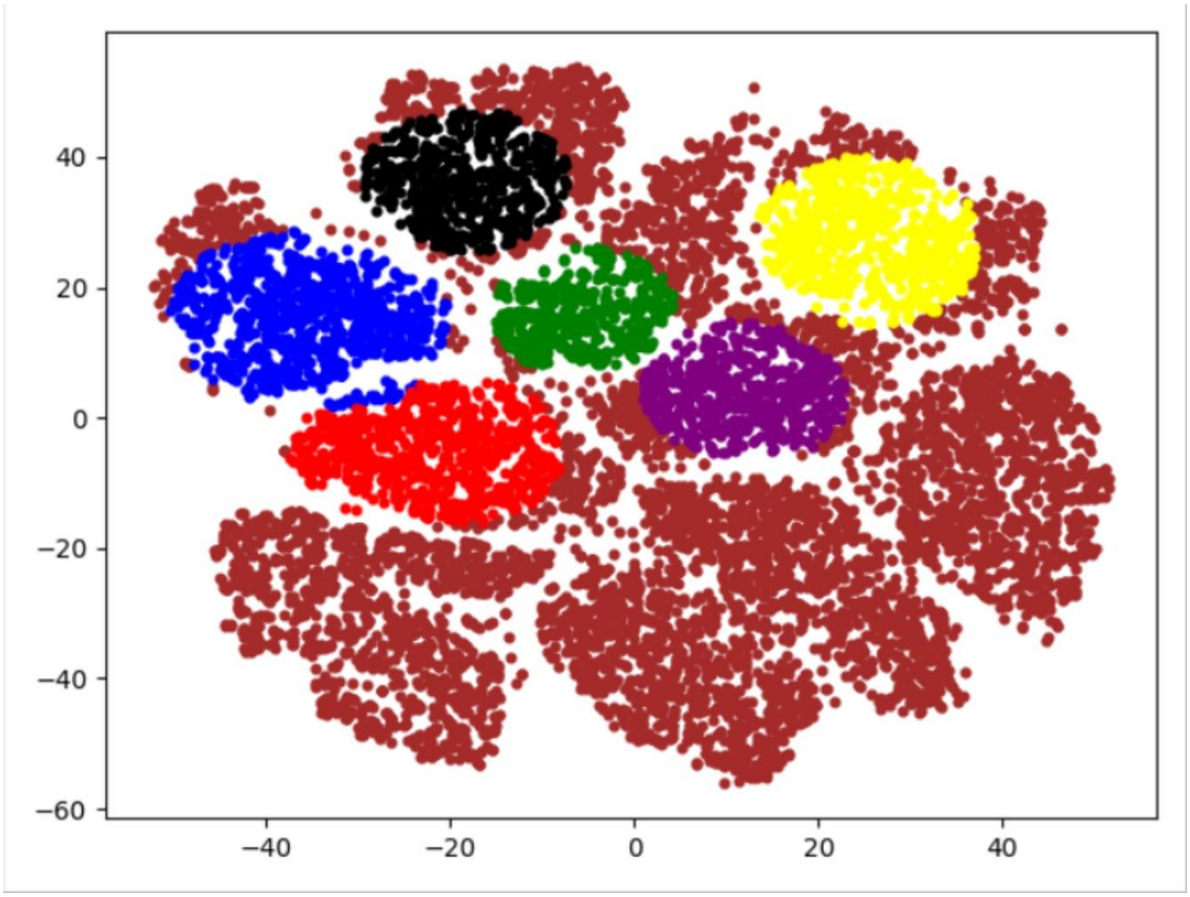
DBSCAN 算法核心过程:

```
# DBSCAN算法核心过程
def  DBSCAN(data,eps,minPts):
    disMat = compute_squared_EDM(data) # 获得距离矩阵
    n, m = data.shape # 数据的行和列

    #   将矩阵中邻域对象个数小于minPts的赋为，大于minPts的赋为0
    # 即遍历找到核心点的集合
    core_points_index = np.where(np.sum(np.where(disMat <= eps, 1, 0), axis=1)
>= minPts)[0]

    labels = np.full((n,), -1) # 初始化所属簇，-1代表未划分
    clusterId = 0 # 簇的个数

    for pointId in core_points_index: # 遍历所有的核心点
        if (labels[pointId] == -1): # 若核心点未划分，将其作为新种子点开始拓展簇集
            labels[pointId] = clusterId # 将当前点所属标记为当前簇

            # 寻找种子点的 Eps邻域中未划分的点，放入待拓展集合
            neighbour=np.where((disMat[:, pointId] <= eps) & (labels==-1))[0]
            seeds = set(neighbour)

            # 以拓展点出发，并标记簇
            while len(seeds) > 0:
                newPoint = seeds.pop()
                labels[newPoint] = clusterId # 标记所属簇

                # 寻找当前拓展点的 Eps 邻域
                queryResults  =  np.where(disMat[:,newPoint]<=eps)[0]

                #  如果新拓展点属于核心点，那么则可以连续密度可达
                if len(queryResults) >= minPts:
                    #  将邻域内未被划分的点加入待拓展集合
                    for resultPoint in queryResults:
                        if labels[resultPoint] == -1:
                            seeds.add(resultPoint)

            clusterId = clusterId + 1 # 生长完毕一个簇
    return labels
```

将分类后的数据可视化显示:

```
def plotFeature(data, labels_):
    clusterNum=len(set(labels_))
    fig = plt.figure()
    scatterColors = ['black', 'blue', 'green', 'yellow', 'red', 'purple',
'orange', 'brown']
    ax = fig.add_subplot(111)
    for i in range(-1,clusterNum):
        colorSytle = scatterColors[i % len(scatterColors)]
        subCluster = data[np.where(labels_==i)]
        ax.scatter(subCluster[:,0], subCluster[:,1], c=colorSytle, s=12)
    plt.show()
```

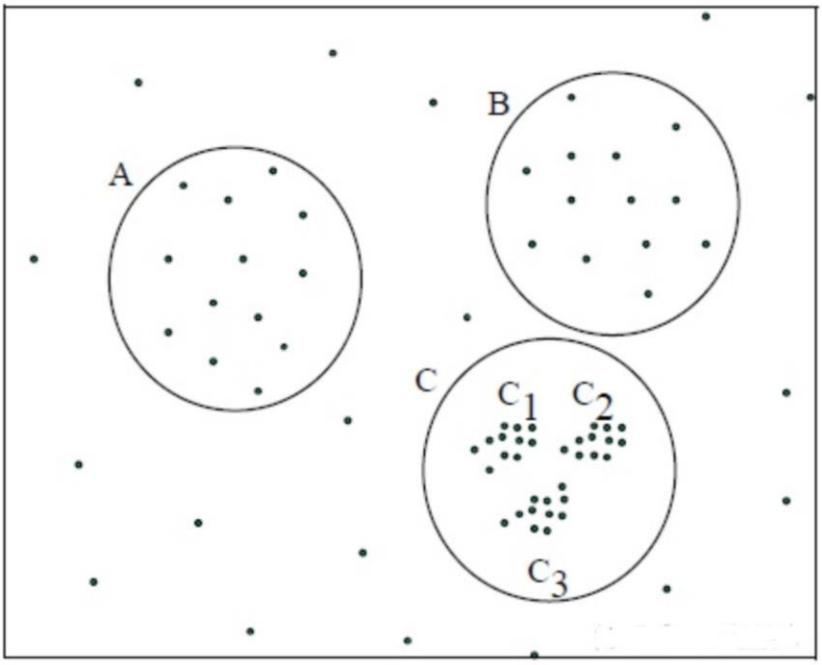
针对本数据集，我们选取了 **Eps=10** 和 **Minpts=500** ，得到聚类结果如下图所示：



由生成的聚类图可知 DBSCAN 算法针对本数据集的聚类结果不理想，原因是 DBSCAN 算法**对输入参数非常敏感**，而参数选取的不同可能会得到不同的聚类结果，部分能形成簇的点被判断为了噪声点。本 算法的设计缺陷在 4.3 小节介绍的 OPTICS 算法中能得到改善。

4.3 OPTICS

DBSCAN 算法对输入参数比较敏感，不合适的参数选取可能导致大量数据被识别为噪声点，因此在输入参数的选取上比较困难，不妨以下图为例：



在 DBSCAN 算法中，选取不同的全局参数可能会得到不同的聚类结果。如上图所示，在我们选取较大的邻域半径 Eps 参数时，会选取 A、B、C 作为三个聚类的结果，判定其他的数据点为噪声点。而当选取的领域半径 Eps 较小时，得到的聚类结果却是 C1、C2、C3 ，此时，A、B 中的点均被判定为噪声点。

因此，选取同一全局参数，不可能同时检测到 A、B、C1、C2、C3。

针对 DBSCAN 算法对输入参数的敏感，OPTICS（Ordering Points To Identify the Clustering Structure）算法使基于密度的聚类结构呈现出一种特殊的顺序，在该顺序下对应的聚类结构包含了每个层级的聚类的信息，并且便于分析。

OPTICS 算法选取了有限个领域参数进行聚类，从而得到不同领域参数下的聚类结果，并由此定义**核心距离（core-distance）** 和**可达距离（reachability-distance）**：

核心距离 $cd(x_i)$:

$$cd(x_i) = \begin{cases} \text{UNDEFINED}, & \text{if}\,|N_\varepsilon(x_i)| < \mathcal{M}; \\ d(x_i, N_\varepsilon^{\mathcal{M}}(x_i)), & \text{otherwise.} \end{cases}$$

其中 $N_\epsilon^i(x_i)$ 表示集合 $N_\epsilon^i(x_i)$ 中与节点 x_i 第 i 最近邻的节点。

不难看出，若 x_i 为核心点，则有 $cd(x_i) < \epsilon$。

可达距离 $rd(x_j, x_i)$:

$$rd(x_j, x_i) = \begin{cases} \text{UNDEFINED}, & \text{if}\,|N_\varepsilon(x_i)| < \mathcal{M}; \\ \max\{cd(x_i), d(x_i, x_j)\}, & \text{otherwise.} \end{cases}$$

当 x_i 为核心点时有

$$rd(x_j, x_i) = \min\{\eta : x_j \in N_\eta(x_i) \text{ and } |N_\eta(x_i)| \geq \mathcal{M}\}$$

即 $rd(x_j, x_i)$ 表示使得 “x_i 为核心点” 且 “x_j 从 x_i 直接密度可达” 同时成立的最小领域半径。

算法步骤：

1、初始时将所有点入队列

2、每次从队列中取出一个对象 i，标记已访问，若该节点为核心点则将 $N_\epsilon(i)$ 中未被访问的节点按照可达距离插入到队列 seedlist 中

3、遍历 seedlist 队列，得到可达距离最小的点 j 并继续拓展，若该节点为核心点则将$N_\epsilon(j)$ 中未被访问的节点按照可达距离插入到队列 seedlist 中，直到队列为空。

4、对于3中生成的可达列表及其距离，依次划分簇，并生成聚类结果。

代码分析：

计算距离矩阵：

```
# 计算距离矩阵
def  compute_squared_EDM(X):
  return  squareform(pdist(X,metric='euclidean'))
```

获得 core_PointID 邻域内满足要求的点集：

```
def
updateSeeds(seeds,core_PointId,neighbours,core_dists,reach_dists,disMat,isProces
s):
    #  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，计算其核心距离
        if(isProcess[neighbour]==-1):
            #  首先计算该点的针对core_PointId的可达距离
            new_reach_dist = max(core_dist, disMat[core_PointId][neighbour])
            if(np.isnan(reach_dists[neighbour])):
                reach_dists[neighbour]=new_reach_dist
                seeds[neighbour] = new_reach_dist
            elif(new_reach_dist<reach_dists[neighbour]):
                reach_dists[neighbour] = new_reach_dist
                seeds[neighbour] = new_reach_dist
    return seeds
```

OPTICS 算法核心流程：

```
def  OPTICS(data,eps=np.inf,minPts=15):
    # 获得距离矩阵
    orders = []
    disMat = compute_squared_EDM(data)

    n, m = data.shape # 数据的行和列
```

```
    #  np.argsort(disMat)[:,minPts-1]  按照距离进行 行排序 找第minPts个元素的索引
    #  disMat[np.arange(0,n),np.argsort(disMat)[:,minPts-1]]  计算minPts个元素的索引的
距离
    temp_core_distances  =  disMat[np.arange(0,n),np.argsort(disMat)[:,minPts-1]]

    # 计算核心距离
    core_dists = np.where(temp_core_distances <= eps, temp_core_distances, -1)

    #  将每一个点的可达距离设为未定义
    reach_dists= np.full((n,), np.nan)

    #  将矩阵的中小于minPts的数赋予1，大于minPts的数赋予零，然后1代表对每一行求和,然后求核心点
坐标的索引
    core_points_index = np.where(np.sum(np.where(disMat <= eps, 1, 0), axis=1)
>= minPts)[0]

    #  用于标识是否被处理，没有被处理，设置为-1
    isProcess = np.full((n,), -1)

    # 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 (isProcess[pointId] == -1):
            # 将点pointId标记为当前类别(即标识为已操作)
            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) & (disMat[:,
pointId] > 0) & (isProcess == -1))[0]
            seeds = dict()

 seeds=updateSeeds(seeds,pointId,neighbours,core_dists,reach_dists,disMat,isProc
ess)

            while  len(seeds)>0:
                nextId  =  sorted(seeds.items(),  key=operator.itemgetter(1))[0][0]
                del  seeds[nextId]
                isProcess[nextId] = 1
                orders.append(nextId)
                #  寻找newPoint种子点eps邻域（包含自己）
                #  这里没有加约束isProcess  ==  -1，是因为如果加了，本是核心点的，可能就变成
了非和核心点
                queryResults = np.where(disMat[:, nextId] <= eps)[0]
                if len(queryResults) >= minPts:

seeds=updateSeeds(seeds,nextId,queryResults,core_dists,reach_dists,disMat,isProc
ess)
                #  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,reach_dists
```

生成聚类结果:

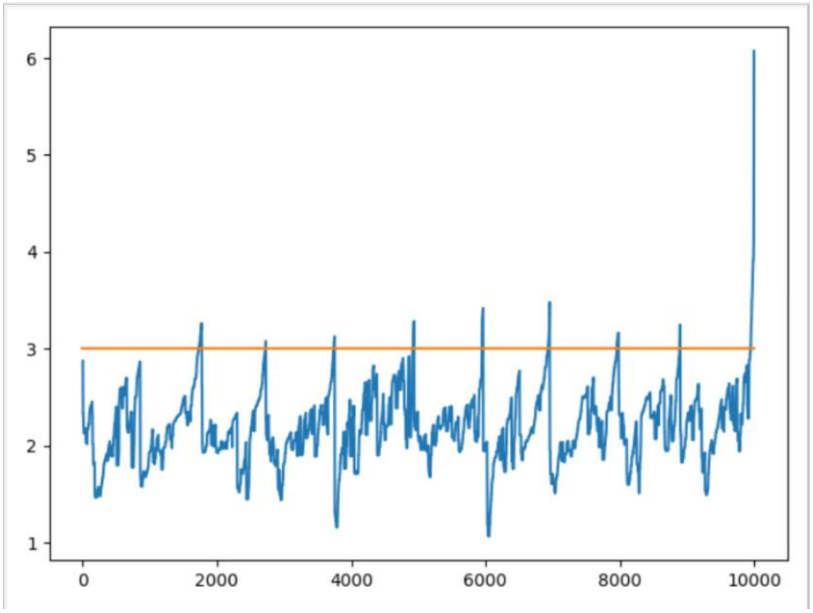
```
def extract_dbscan(data,orders, reach_dists, eps):
    n,m=data.shape # 原始数据的行和列
```

```
    #  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]，找到有序列表中小于eps的点的索引
    reach_distIds=np.where(reach_dists[orders] <= eps)[0]

    #   正常来说：current的值的值应该比pre的值多一个索引。如果大于一个索引就说明不是一个类别
    pre=reach_distIds[0]-1
    clusterId=0
    labels=np.full((n,),-1)

    for current in reach_distIds:
        #   正常来说：current的值的值应该比pre的值多一个索引。如果大于一个索引就说明不是一个类
别
        if(current-pre!=1):
            # 类别+1
            clusterId=clusterId+1
        labels[orders[current]]=clusterId
        pre=current
    return labels
```

显示决策图：

```
# 显示决策图
def plotReachability(data,eps):
    plt.figure()
    plt.plot(range(0,len(data)), data)
    plt.plot([0, len(data)], [eps, eps])
    plt.show()
```

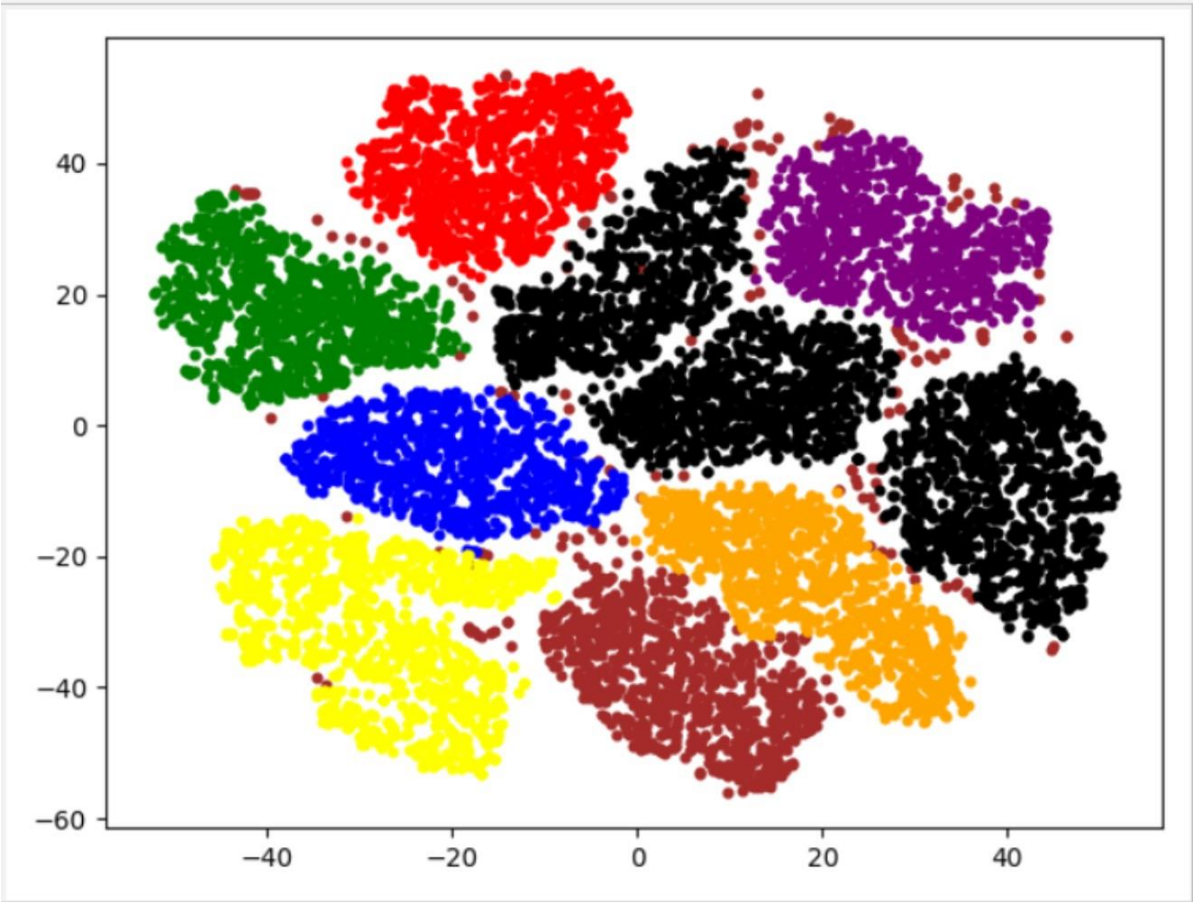
OPTICS 算法针对本数据集的决策图如下：



显示分类的类别：

```
# 显示分类的类别
def plotFeature(data,labels):
    clusterNum = len(set(labels))
    fig = plt.figure()
    scatterColors = ['black', 'blue', 'green', 'yellow', 'red', 'purple',
'orange', 'brown']
    ax = fig.add_subplot(111)
    for i in range(-1, clusterNum):
        colorSytle = scatterColors[i % len(scatterColors)]
        subCluster = data[np.where(labels == i)]
        ax.scatter(subCluster[:, 0], subCluster[:, 1], c=colorSytle, s=12)
    plt.show()
```

OPTICS 算法生成的聚类结果如下图所示：



由生成的聚类图可知 OPTICS 算法的聚类效果在 DBSCAN 算法的基础上有了很大改善。

4.4 CFSFDP 基于密度峰值的快速聚类算法

相较于其他的聚类算法，CFSFDP 克服了很多困难，例如 K-means 无法应对非球状簇， DBSCAN 需要设置密度阈值，阈值对聚类效果具有很大的影响但是调参过程会花费很多精力。CFSFDP 通过**同时考虑数据点分布密度和高密度数据点之间的距离**，达到克服如上问题的效果。

该算法定义聚类中心的规则为：**聚类中心**是一定范围内数据点密度最高的点；聚类中心与其他比其密度更高的点之间的距离更远。

这里距离的度量采用欧式距离：

$$dist\left(x_i, x_i\right)=\sqrt{\sum_{d=1}^{D}\left(x_i^d-x_j^d\right)^2}$$

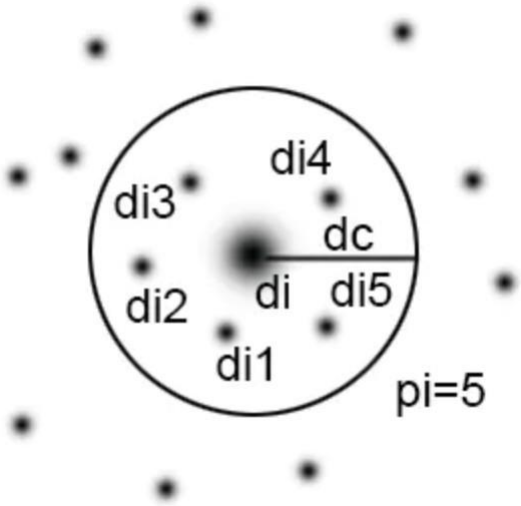
密度的定义为：

$$\rho_i=\sum_{x_j \in U} \chi\left(dist\left(x_i, x_j\right)-dist_{cutoff}\right)$$

其中：

$$\chi(x)=\begin{cases}1 & x \leq 0 \\ 0 & x>0\end{cases}$$

直观表示为在该点附近一定范围内（阈值，超参数）的点数。



常规的聚类算法仅仅要求到聚类中心周围数据点密度高即可，但本算法还引入了高密度点之间距离的约束，即高密度点之间的距离需要尽可能大。其中靠如下变量来约束，代表密度大于本身的点到该点距离的最小值，如果是聚类中心则需要使该值尽可能大。

$$\delta_i=\min_{j:\rho_j>\rho_i}\left(dist(x_i, x_j)\right)$$

聚类代码如下：

CFSFDP 算法具体逻辑：

```
def CFSFDP(data,dc):
    n,m=data.shape
    #  制作任意两点之间的距离矩阵
    disMat  =  squareform(pdist(data,metric='euclidean'))
    # 计算每一个点的密度（在dc的园中包含几个点）
    densityArr = np.where(disMat < dc, 1, 0).sum(axis=1)
    # 将数据点按照密度大小进行排序（从小到大）
    densitySortArr=np.argsort(densityArr)
    # 初始化，比自己密度大的且最近的距离
    closestDisOverSelfDensity  =  np.zeros((n,))
    #  初始化，比自己密度大的且最近的距离对应的节点id
    closestNodeIdArr = np.zeros((n,), dtype=np.int32)
    # 从密度最小的点开始遍历
    for index,nodeId in enumerate(densitySortArr):
```

```
        #  点密度大于当前点的点集合
        nodeIdArr = densitySortArr[index+1:]
        # 如果不是密度最大的点
        if nodeIdArr.size != 0:
            # 计算比自己密度大的点距离nodeId的距离集合
            largerDistArr = disMat[nodeId][nodeIdArr]
            # 寻找到比自己密度大，且最小的距离节点
            closestDisOverSelfDensity[nodeId] = np.min(largerDistArr)
            #   寻找到最小值的索引，索引实在largerdist里面的索引（确保是比nodeId）节点大
            #  如果存在多个最近的节点，取第一个
            #  注意，这里是largerDistArr里面的索引
            min_distance_index = np.argwhere(largerDistArr ==
closestDisOverSelfDensity[nodeId])[0][0]
            # 获得整个数据中的索引值
            closestNodeIdArr[nodeId] = nodeIdArr[min_distance_index]else:
            #   如果是密度最大的点，距离设置为最大，且其对应的ID设置为本身
            closestDisOverSelfDensity[nodeId] =
np.max(closestDisOverSelfDensity)
            closestNodeIdArr[nodeId] = nodeId
    #    由于密度和最短距离两个属性的数量级可能不一样，分别对两者做归一化使结果更平滑
    normal_den = (densityArr - np.min(densityArr)) / (np.max(densityArr) -
np.min(densityArr))
    normal_dis = (closestDisOverSelfDensity - np.min(closestDisOverSelfDensity))
/ (
                np.max(closestDisOverSelfDensity) -
np.min(closestDisOverSelfDensity))
    gamma = normal_den * normal_dis
    #  densityArr：里面保存了每一个点的密度大小
    #  densitySortArr：数组中保存了每一个点排序
    #   closestDisOverSelfDensity：数组保存了比本身密度大，且最近的距离
    # closestNodeIdArr:数组保存了比本身密度大，且最近的节点id
    #  gamma:每个点的评分(综合距离和密度)
    return
densityArr,densitySortArr,closestDisOverSelfDensity,closestNodeIdArr,gamma
```

密度与距离图显示函数：

```
def showDenDisAndDataSet(den, dis, ds):
    plt.figure(num=1, figsize=(15, 9))
    ax1 = plt.subplot(121)
    plt.scatter(x=den, y=dis, c='k', marker='o', s=15)
    plt.xlabel('密度')
    plt.ylabel('距离')
    plt.title('决策图')
    plt.sca(ax1)
    # 将数据集显示在图形面板
    ax2 = plt.subplot(122)
    plt.scatter(x=ds[:, 0], y=ds[:, 1], marker='o', c='k', s=8)
    plt.xlabel('经度')
    plt.ylabel('纬度')
    plt.title('数据集')
    plt.sca(ax2)
    plt.show()
```

确定类别点,计算每点的密度值与最小距离值的乘积，并画出决策图，以供选择将数据共分为几个类别:

```
def show_nodes_for_chosing_mainly_leaders(gamma):
    plt.figure(num=2, figsize=(15, 10))
    # -np.sort(-gamma) 将gamma从大到小排序
    plt.scatter(x=range(len(gamma)), y=-np.sort(-gamma), c='k', marker='o', s=-
np.sort(-gamma) * 100)
    plt.xlabel('点的数量')
    plt.ylabel('每个点的评分')
    plt.title('递减顺序排列的γ')
    plt.show()
```

最终聚类图绘制:

```
def show_result(labels, data, corePoints):
    plt.figure(num=3, figsize=(15, 10))
    # 一共有多少类别
    clusterNum = len(set(labels))
    scatterColors = [
              '#FF0000', '#FFA500', '#FFFF00', '#00FF00', '#228B22',
              '#0000FF', '#FF1493', '#EE82EE', '#000000', '#106422',
              '#00FF00', '#006400', '#00FFFF', '#0000FF', '#FFFACD',
    ]
    # 绘制分类数据
    for i in range(clusterNum):
        # 为i类别选择颜色
        colorSytle = scatterColors[i % len(scatterColors)]
        # 选择该类别的所有Node
        subCluster = data[np.where(labels == i)]
        plt.scatter(subCluster[:, 0], subCluster[:, 1], c=colorSytle, s=5,
marker='o', alpha=0.66)
    # 绘制每一个类别的聚类中心
    plt.scatter(x=data[corePoints, 0], y=data[corePoints, 1], marker='+', s=100,
c='K', alpha=0.8)
    plt.title('聚类结果图')
    plt.show()
```

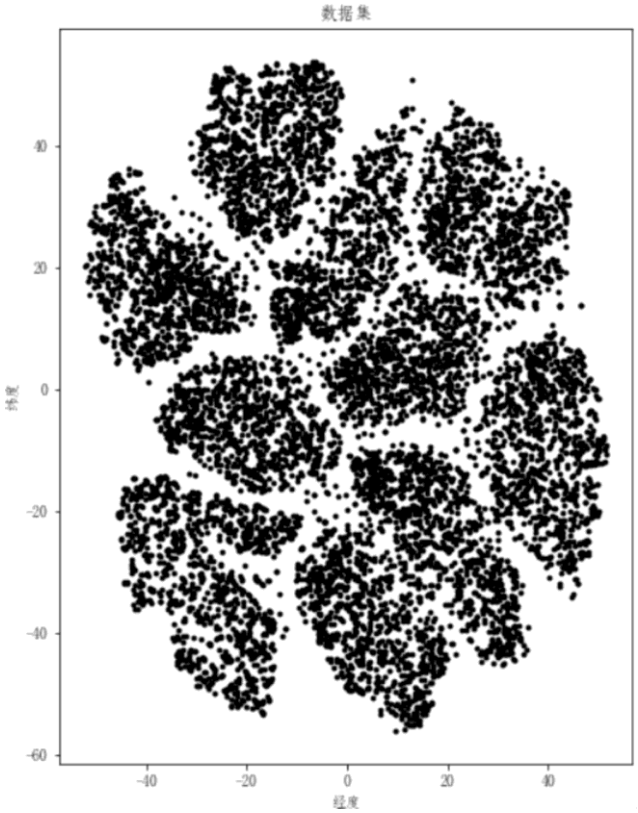
确定每点的最终分类:

```
def extract_cluster(densitySortArr,closestNodeIdArr, classNum,gamma):
    n=densitySortArr.shape[0]
    # 初始化每一个点的类别
    labels=np.full((n,),-1)
    corePoints =  np.argsort(-gamma)[: classNum]  # 选择
    # 将选择的聚类中心赋予类别
    labels[corePoints]=range(len(corePoints))
    # 将ndarrar数组转为list集合
    densitySortList=densitySortArr.tolist()
    # 将集合元素反转，即密度从大到小的排序索引
    densitySortList.reverse()
    #  循环赋值每一个元素的label
```

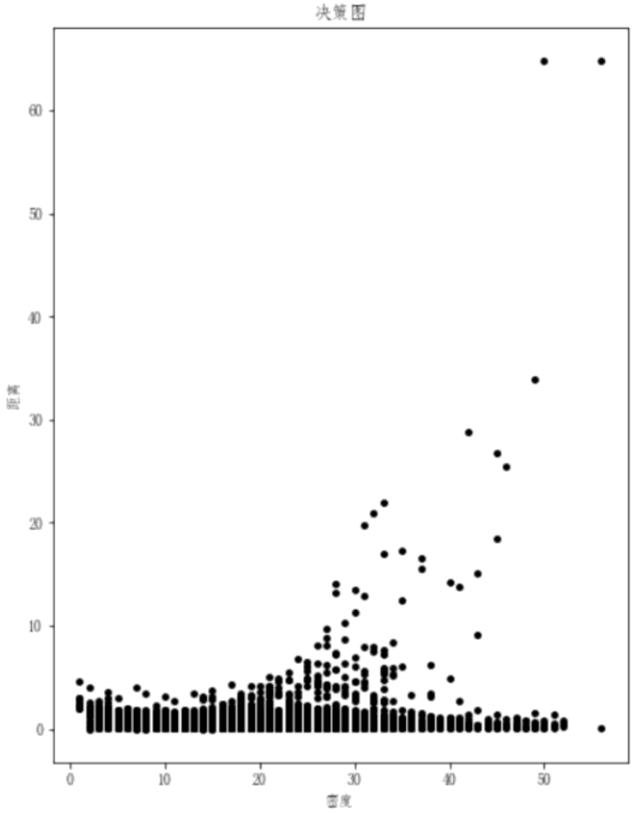
```
    for nodeId in densitySortList:
        if(labels[nodeId]==-1):
            # 如果nodeId节点没有类别
            #  首先获得closestNodeIdArr[nodeId]  比自己密度大，且距离自己最近的点的索引
            # 将比自己密度大，且距离自己最近的点的类别复制给nodeId
            labels[nodeId]=labels[closestNodeIdArr[nodeId]]
    return  corePoints,labels
```

聚类过程：

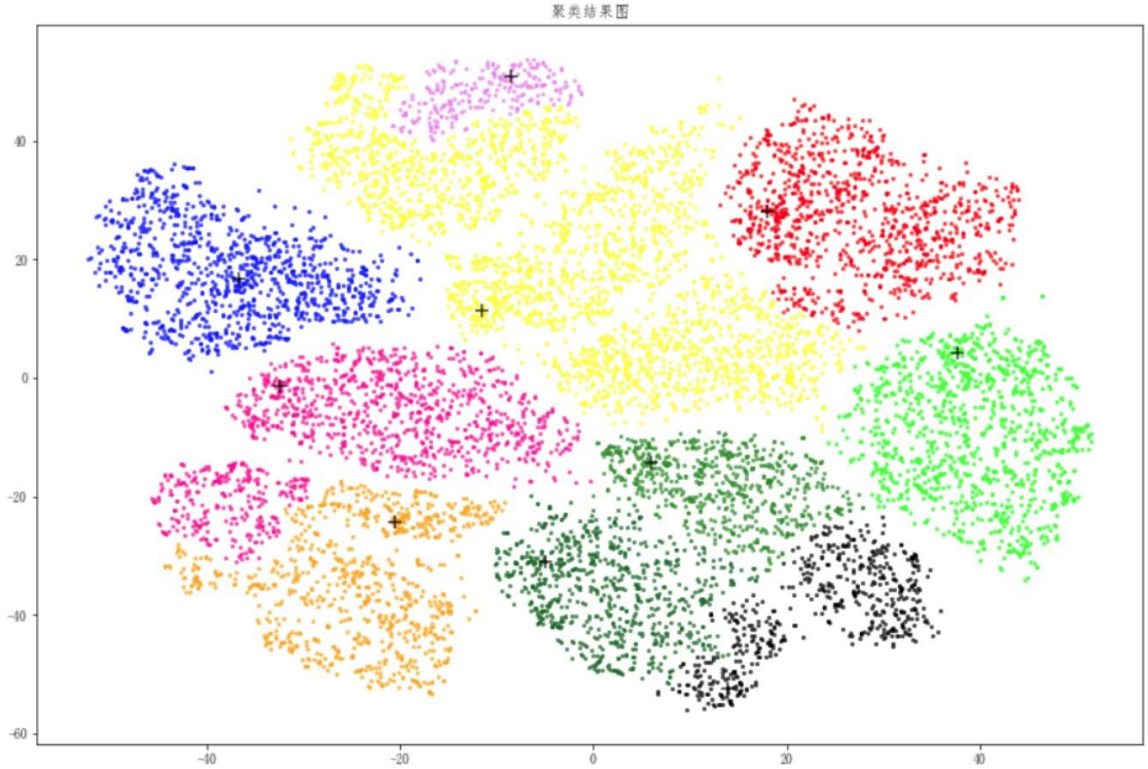
首先获取数据，打印出当前数据的分布：



通过计算得到决策图，决策图横坐标为密度，纵坐标为与其他高密度点的距离，我们需要寻找的聚类中心就是这两个指标同时较大的点，出现在右上角：



输入聚类的点数n可以选择最符合标准的n个点作为聚类中心，之后生成的聚类图如下所示：



可以看到该图中聚类结果比较一般，原因是数据分布密度并不完全反应中心关系，很多在同一类的图片在较大尺度上形成了聚类，但在小尺度上的密度并不集中，最后导致中心偏移。但是本算法在设计上针对上文阐述的问题还是引入了很好的考虑。

4.5 AP聚类算法

AP 算法是基于数据点间的“信息传递的聚类算法”，与前面大多数算法不同的是，AP聚类算法并不需要指定聚类的个数，它通过在样本之间**传播吸引度和归属度**的方式，来产生 m 个高质量的质心，并将所有点归结在这个质心中。在整个迭代过程中，声明了两个矩阵：

- 代表矩阵——R ，其第i行k列的元素描述了数据对象k适合作为数据对象i聚类中心的程度。
- 归属矩阵——A ：其第i行k列的元素描述了数据对象i选择数据对象k作为其聚类中心的适合程度。

在每轮的迭代过程中，两个**矩阵的更新策略**如下所示：

R的更新方式：

$$r_{t+1}(i,k) = s(i,k) - \max_{k' \neq k}\{a_t(i,k') + s(i,k')\}$$

A的更新方式：

$$a_{t+1}(i,k) = \min\left(0, r_t(k,k) + \sum_{i' \notin \{i,k\}} \max\{0, r_t(i',k)\}\right), i \neq k$$

和

$$a_{t+1}(k,k) = \sum_{i' \neq k} \max\{0, r_t(i',k)\}$$

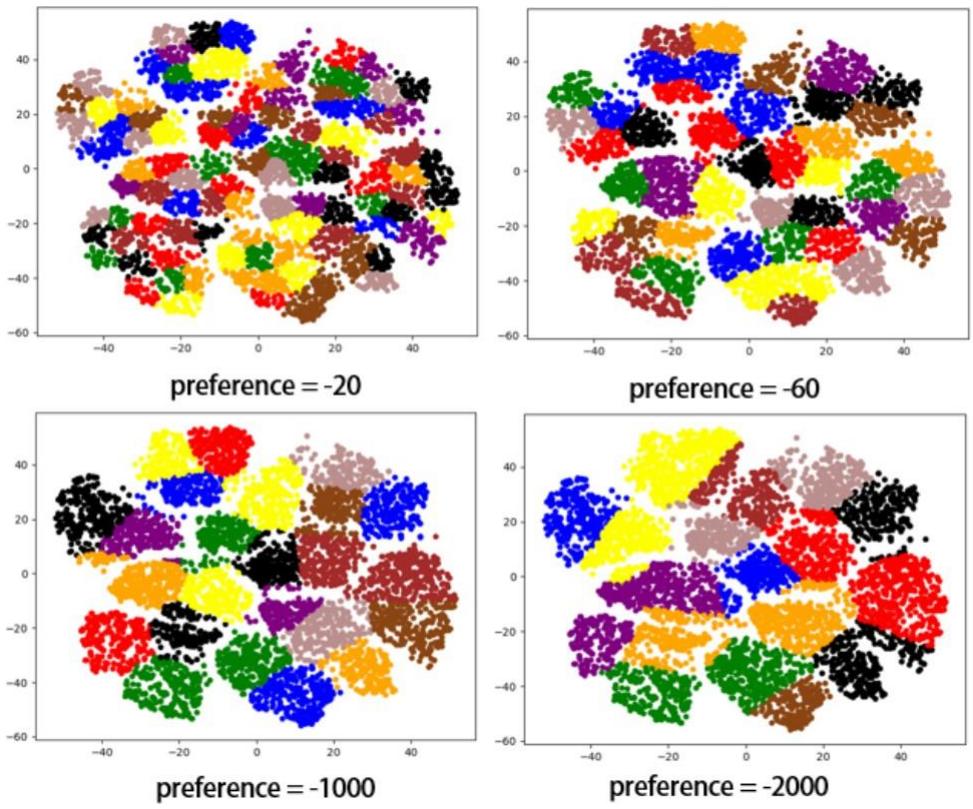
其中t是迭代的轮数

进行聚类的函数展示如下：

```
# data:进行聚类的样本点集
#  preference：数据点的参考度
#  damping：阻尼系数，用于控制聚类的收敛速度
# max_iter：最大迭代次数
#  convergence_iter:指定多少次聚类中心不变后停止迭代
# similarity:相似度，可以指定，也可以自动计算
def
MYAffinityPropagation(data,preference,damping,max_iter,convergence_iter,similari
ty = None,precomputed=False):
    n,m = data.shape
    if precomputed == False:
        similarity = - squareform(pdist(data,metric='euclidean'))
    similarity.flat[::(n+1)] = preference
    stop = np.zeros((n, convergence_iter))
    # 初始化代表矩阵
    responsibility = np.zeros((n, n))
    # 初始化归属矩阵
    availability = np.zeros((n, n))
    ind = np.arange(n)
    for idx in tqdm.tqdm(range(max_iter)):
        # 迭代更新归属矩阵
        a_add_s = np.add(availability, similarity)
        maxIdx,maxY = np.argmax(a_add_s, axis=1), np.max(a_add_s, axis=1)
        a_add_s[ind, maxIdx] = - np.inf
        maxSecondY = np.max(a_add_s, axis=1)
```

```
            oldResponsibility = similarity - np.tile(maxY.reshape(-1,1), (1,n))
            oldResponsibility[ind, maxIdx] = similarity[ind, maxIdx] - maxSecondY

            # 迭代更新代表矩阵
            responsibility = responsibility*damping + (1-damping)*oldResponsibility
            zero_max_r = np.maximum(responsibility, 0)
            zero_max_r.flat[::n + 1] = responsibility.flat[::n + 1]
            zero_max_r = np.sum(zero_max_r, axis=0)-zero_max_r
            dA = np.diag(zero_max_r).copy()
            oldAvailability = np.minimum(0, zero_max_r)
            oldAvailability.flat[::(n + 1)] = dA
            availability = availability * damping + (1 - damping) * oldAvailability
            E = (np.diag(availability) + np.diag(responsibility)) > 0
            stop[:, idx % convergence_iter] = E
            K = np.sum(E, axis=0)
            #  当迭代次数大于convergence_iter，才有可能存在停止的可能性
            if idx >= convergence_iter:
                se = np.sum(stop, axis=1)
                converged = np.sum((se == convergence_iter) + (se == 0)) == n
                if converged and (K > 0):
                    break
    exemplars = np.where(np.diag(availability + responsibility) > 0)[0]
    # 返回exemplars的索引
    return  exemplars,similarity

def  extract_cluster(exemplars,  similarity):
    # 距离哪一个点相似度大，就属于哪一个类别
    labels  =  np.argmax(similarity[:,exemplars],axis=1)
    for index,exemplar in enumerate(exemplars):
        labels[exemplar]=index
    return labels
```

基于上述代码，对样本点尝试进行聚类，并得到下述实验结果：



如上图所示，虽然如前文所述，AP算法并未指定聚类的个数，但是数据点的参考度对于结果有着巨大的影响，实际上也就控制着最后聚类的结果。虽然AP算法在我们的图像聚类任务上表现得并不太好，但是对于事先并不知道聚类数量的聚类任务，AP算法显示了巨大的潜力。

五、聚类结果评估分析

5.1 轮廓系数

轮廓系数（Silhouette Coefficient），是聚类效果好坏的一种评价方式。 它结合**内聚度**和**分离度**两种因素。可以用来在相同原始数据的基础上用来评价不同算法、或者算法不同运行方式对聚类结果所产生的影响。轮廓系数**取值范围为[-1,1]**，取值越接近1则说明聚类性能越好，相反，取值越接近-1则说明聚类性能越差。

若定义：

- a：某个样本与其所在簇内其他样本的平均距离
- b：某个样本与其他簇样本的平均距离

  则有某个样本的轮廓系数为：

$$s = \frac{b-a}{max(a,b)}$$

总的轮廓系数为：

$$SC = \frac{1}{N}\sum_{i=1}^{N} s_i$$

我们采用轮廓系数对第四部分所使用的聚类算法进行评估，有如下的结果：

算法K-Means	DBSCAN		OPTICS	CFSFDP	AP
轮廓系数0.384	-0.013		0.371	0.291	0.374

5.2 兰德系数

兰德系数（Rand Index）是一种较为简单的评价聚类结果好坏的方法。与轮廓系数不同的是，兰德系数在计算的时候需要知道**外部评价标准**（true_label）。兰德系数的值在[0,1]之间，当聚类结果完美匹配时，兰德系数为1。

给定 n个对象集合$S = \{O_1, O_2, O_3 \ldots, O_n\}$，且有$U = \{u_1, u_2, u_3 \ldots, u_r\}$和$V = \{v_1, v_2, v_3 \cdot \ldots, v_r\}$，其中$U$为外部评价标准，$V$为聚类结果。

设定四个统计量：

- a为在U中为同一类且在V中也为同一类别的数据点对数
- b为在U中为同一类但在V中却隶属于不同类别的数据点对数

- c为在U中不在同一类但在V中为同一类别的数据点对数
- d为在U中不在同一类且在V中也不属于同一类别的数据点对数

此时兰德系数为：

$$RI = \frac{a+d}{a+b+c+d}$$

我们采用兰德系数对第四部分所使用的聚类算法进行评估，有如下的结果：

算法K-Means	DBSCAN		OPTICS	CFSFDP	AP
兰德系数0.818	0.594		0.810	0.794	0.932

5.3 总述与聚类算法选择

通过聚类的**可视化结果**和聚类的相应**评价指标**的对比，我们发现最基础的聚类算法 K-means 的表现最佳，原因如下：

- 首先，在针对本数据集求解聚类结果时，若我们预先可知数据集可划分为 10 个类别，则这一可用信息可以对 K-means 算法的聚类分析提供极大帮助。这一重要信息在其他一些自行确定种类数的算法中并未得到合理利用，从而出现 类别判断错误导致聚类与理想状态相差较大 的情况。
- 其次，经过 t-SNE 算法降维后，各类数据的聚集状态基本为球状，是 K-means 可以解决的聚类状态，并不存在明显的缺陷。
- 最后，由于在二维空间中特征集散状态相对均匀，初始化的随机性对于K-means的影响不会太大，而经过几个初始化选择的试探，我们可以找到一个相对较好的初始化来形成聚类，可以得到较好的结果。