目录

算法实现流程：

算法主要使用 python 完成，实现调用的外部包包括 numpy, pandas, matplotlib。

1. 读取数据

```python
def readData(fileName):
    csvData = pd.read_csv(fileName,header=None)  # 读取训练数据
    # csvData.plot.scatter(x=0, y=1)
    return csvData
```

2. 计算距离矩阵

```python
def computeDistance(item1,item2):
    res = 0
    for i in range(len(item1)):
        res += (item1[i]-item2[i])**2
    return np.sqrt(res)
    # return res

# 计算距离矩阵
def distanceMatrix(rawdata,disFun=computeDistance):
    l = len(rawdata)
    res = [[0 for _ in range(l)] for _ in range(l)]
    disList = []
    for i in range(l):
        for j in range(i+1,l):
            tmp = disFun(rawdata.iloc[i],rawdata.iloc[j])
            res[i][j] = res[j][i] = tmp
            disList.append(tmp)
    disList.sort()
    return pd.DataFrame(res),pd.Series(disList)
```

3. 计算局部密度与最近距离

```python
# 直接确定dc,高斯核函数计算密度
def gs_computeDensity(dmatrix,dlist,dcPercent=0.02):
    l = len(dmatrix)
    position = int(len(dlist)*dcPercent)
    dc = dlist.iloc[position,0]
    density = pd.Series([0.0 for _ in range(l)])
    for i in range(l):
        tmp = dmatrix.iloc[i]
        # -1 是为了减去矩阵中对角线上的0元素产生的1
        density[i] = tmp.map(lambda x: np.exp(-(x/dc)*(x/dc))).sum()-1
    averageNeighbors = density.mean()

    print("dc found: " + str(dc))
    print("averageNeighbors: " + str(averageNeighbors))
    return density,dc

# compute min distance to higher density
def computeMDTH(dmatrix,density):
    l = len(dmatrix)
    MDTH = [0 for _ in range(l)]
    for i in range(l):
        currentDistance = dmatrix.iloc[i,:]
        qualifiedIndex = density[(density >= density[i]) & (density.index != i) ].index
        qualifiedDistance = currentDistance[qualifiedIndex]
        if len(qualifiedDistance) == 0:
            MDTH[i] = currentDistance.max()
        else:
            MDTH[i] = qualifiedDistance.min()
    return pd.Series(MDTH)
```

4. 确定聚类中心

```python
# 确定聚类中心、生成聚类结果
clusters = (density*(mdth**2)).sort_values(ascending=False)[:7].index
assignTag(dmatrix,rawData,clusters)
findNoise(dmatrix,density,rawData,clusters,dc)
```

5. 生成聚类结果

```python
def assignTag(dmatrix,rawData,clusters):
    l = len(dmatrix)
    tag = [0 for _ in range(l)]
    count = 0
    id2tag = {}
    for i in range(l):
        currentDistance = dmatrix.iloc[i,:]
        distances2Centers = currentDistance[clusters]
        tmp = distances2Centers.idxmin()
        if not tmp in id2tag:
            id2tag[tmp] = count
            count += 1
        tag[i] = id2tag[tmp]
    rawData['tag'] = tag
```

6. 区分出 cluster core 与 cluster halo 区域

```
def findNoise(dmatrix,density,rawData,clusters,dc):
    for i in clusters:
        clusterIndex = i
        clusterMem = rawData[rawData['tag'] == clusterIndex].index
        border = []
        for mem in clusterMem:
            currentDistance = dmatrix.iloc[mem]
            for i in range(len(currentDistance)):
                if not i in clusterMem and currentDistance[i] < dc:
                    border.append(mem)
                    break
        maxBorderDensity = density[border].max()
        for mem in clusterMem:
            if density[mem] <= maxBorderDensity:
                rawData.ix[mem,'tag'] = -1
```

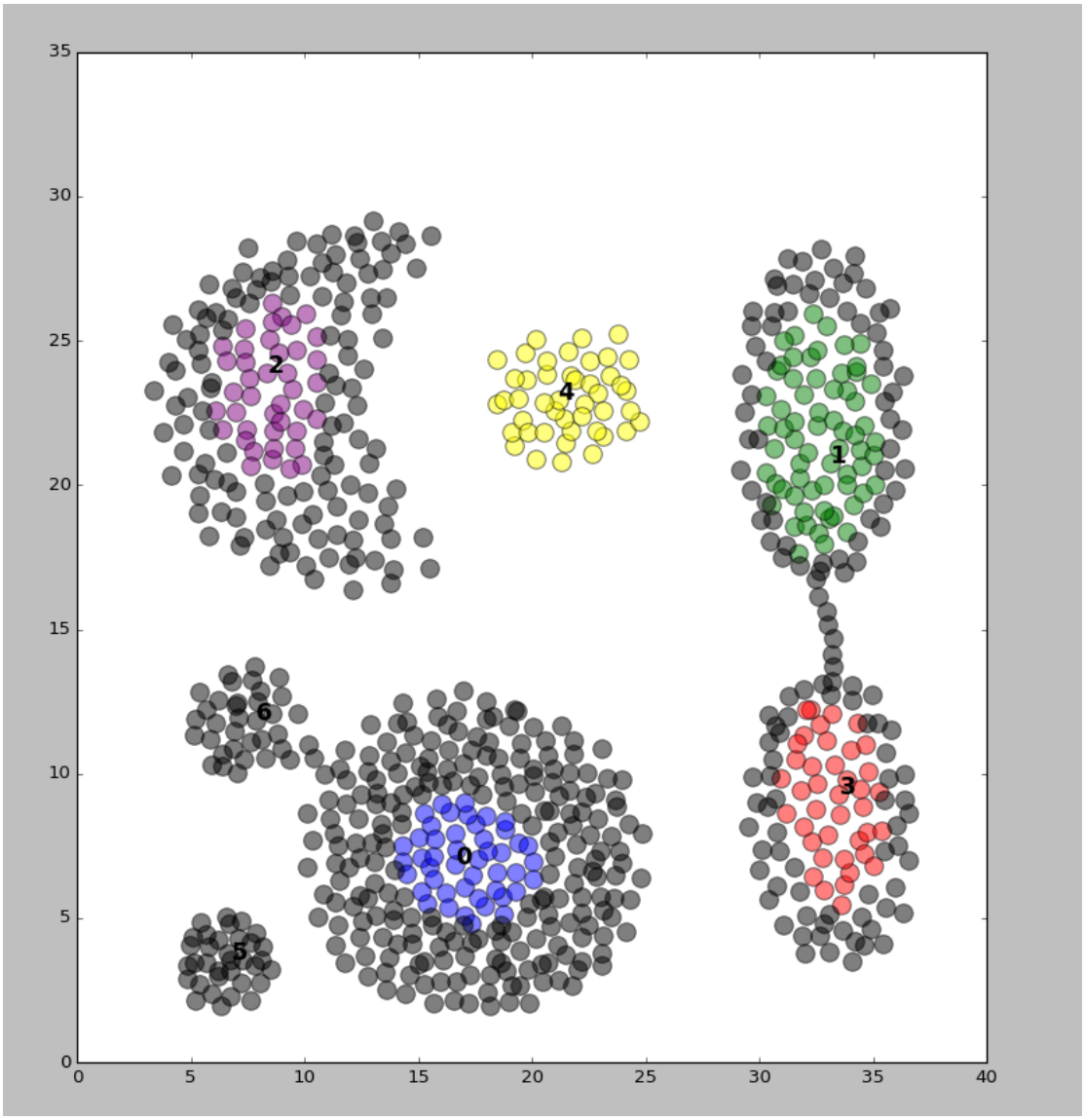
7. 结果可视化

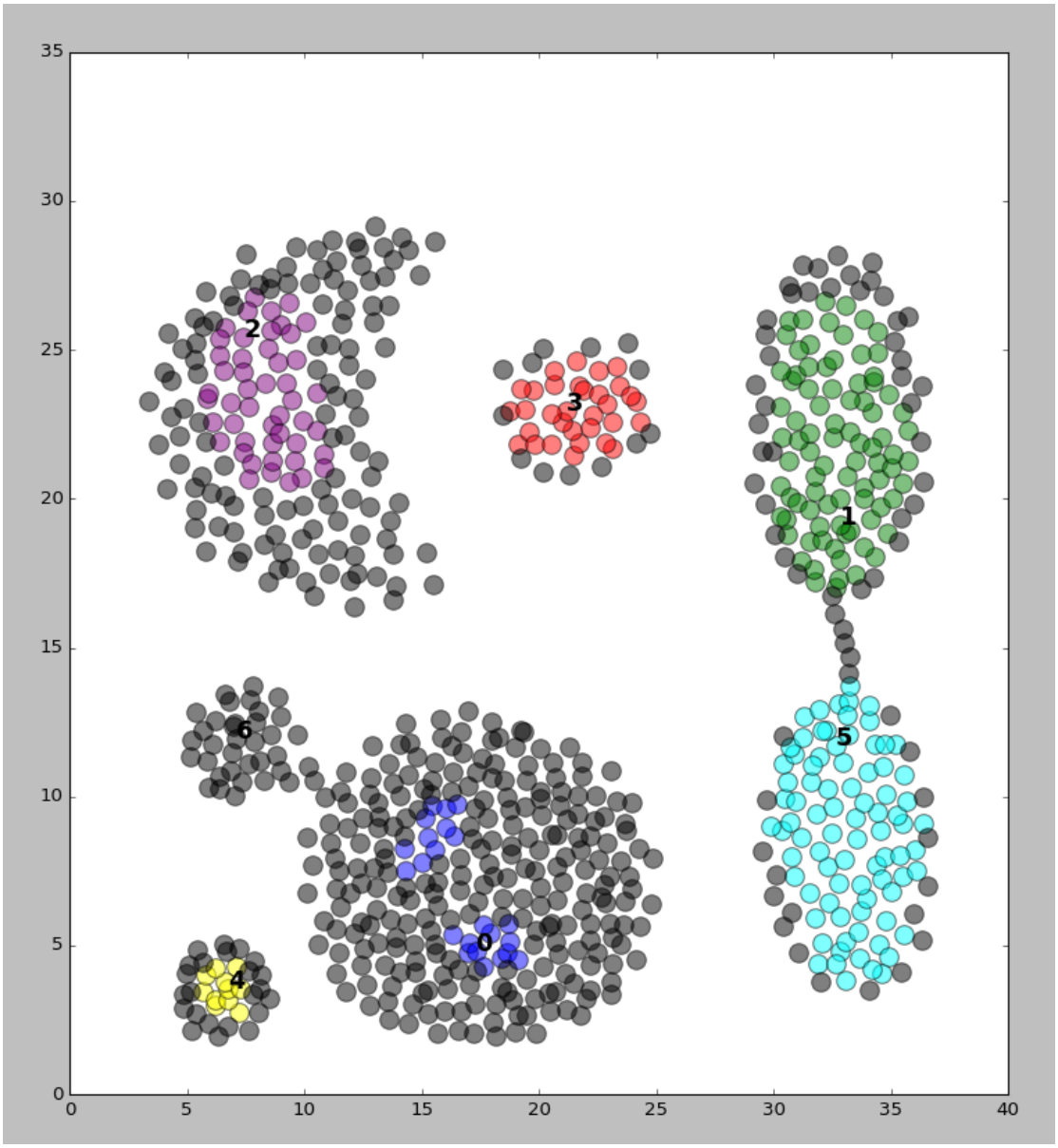
```
def plotResult(density,mdth,rawData):
    f,(ax11, ax12) = plt.subplots(1, 2)

    # ax12.scatter(rawData[0],rawData[1],s=density*5)
    # ax11.scatter((density*mdth).index,(density*mdth).sort_values())
    ax12.scatter(density,mdth)
    colors = ['blue','green','purple','red','yellow','cyan','m']
    # cluster core
    for i in range(7):
        index = clusters[i]
        x = rawData[rawData['tag']==index][0]
        y = rawData[rawData['tag']==index][1]
        ax11.scatter(x,y,c=colors[i], alpha=0.5,s = 150)
        ax11.annotate(i,(rawData[0][index],rawData[1][index]), fontsize=16,fontweight="bold")
        ax12.annotate(i,(density[index],mdth[index]), fontsize=16)
    # cluster halo
    noise = rawData[rawData['tag']==-1]
    ax11.scatter(noise[0],noise[1],c='black', alpha=0.5,s = 150)
    plt.show()
```

结果展示：

下图展示的是聚类的结果可视化展示。图中使用不同的颜色区分不同的聚类，黑色的点表示该点属于 cluster halo 疑似噪音。数字标识的点是算法选取的聚类中心的位置。


35
30
25
20
15
10
5
0
0
5
10
15
20
25
30
35
40
2
4
1
6
0
3
5



算法关键点：

1. 参数的选取问题

该算法的主要可以调整的参数有两个。一个是 dc，dc 会直接影响局部密度的计算过程。另一个参数是聚类的数目。这个参数其实是隐含在算法里的，而且它并不能随意指定，而要根据 decision tree 的结果来随机应变。

1）dc 的选取
该算法 dc 一个参数。论文的说法是结果对 dc 参数选取并不敏感。但是根据我的实验结果，dc 选取还是挺玄学的；最终结果的好坏与 dc 的选取有极大的联系。那么如何确定最优的 dc 呢？论文里给出一个相对的评价标准。

> Varying d_c for the data in Fig. 2B produced mutually consistent results (fig. S1). As a rule of thumb, one can choose d_c so that the average number of neighbors is around 1 to 2% of the total number of points in the data set. For data

翻译过来就是选取的 dc 使每个点的平均邻居数目占所有数据点的 1%到 2%。
现在问题就变成了怎么使选取的 dc 达到这样的标准。

首先，我注意到的一点是 dc 与点的平均邻居数目是成正比的。也就是说 dc 越大，平均邻居越多。那其实选取合适的 dc 这个问题可以转化为，一个在虚拟的有序数组（把 dc 看作索引、把邻居数目看作值）里查找某个符合标准的数的问题。可以用二分查找来解决。即只需设置一个 dc 的上界（我是把其设为随机某维的数组中位数），把 0 看作 dc 的下界，然后设置一个 dc 选取指标范围，一定可以找到一个符合标准的 dc 的值。

但是这个方法的弊端显而易见——需要多次迭代才能找到合适的 dc 值。于是这就引出了另一种确定 dc 的方法。这种方法直接指定一个百分比的指标（平均邻居数目占所有数据点的），计算出对应的 dc 值。它的做法也很简单，把点之间的距离保存在数组里，排序，选出百分比对应的索引位置上的值作为 dc 的值。在使用截断距离作为局部密度计算标准的场景下，它的正确性是可以经过数学方法证明的（这里就不再证明了）。

2）聚类数目的选取

这个参数实际作用实际是在确定聚类中心的个数。根据论文，这个过程是根据数据在 mdth 和 local density 两个维度的分布（decision tree）来确定的。但是在算法运行过程中，如何让计算机发现聚类中心呢？论文里的一个示例，是直接使用 mdth*density 观察图的拐点。但实际无论是找离群点，还是拐点都是一个想当主观的过程。

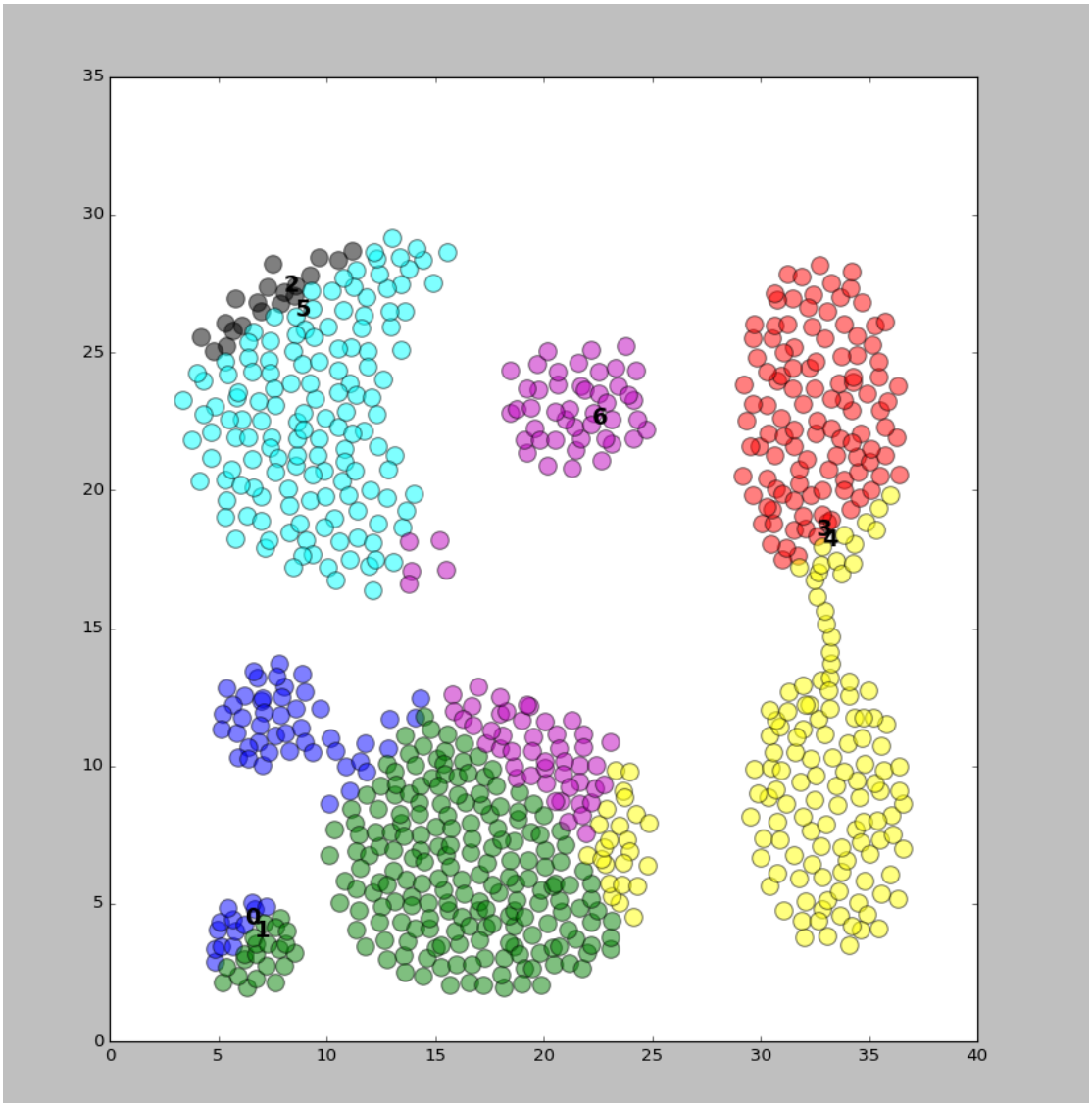
这个问题上我也没有找到很合适的解决办法，只能还是通过观察 decision tree 手动选择合适的聚类数目。

2. MDTH（the minimum distance between the point i and any other point with higher density）的计算

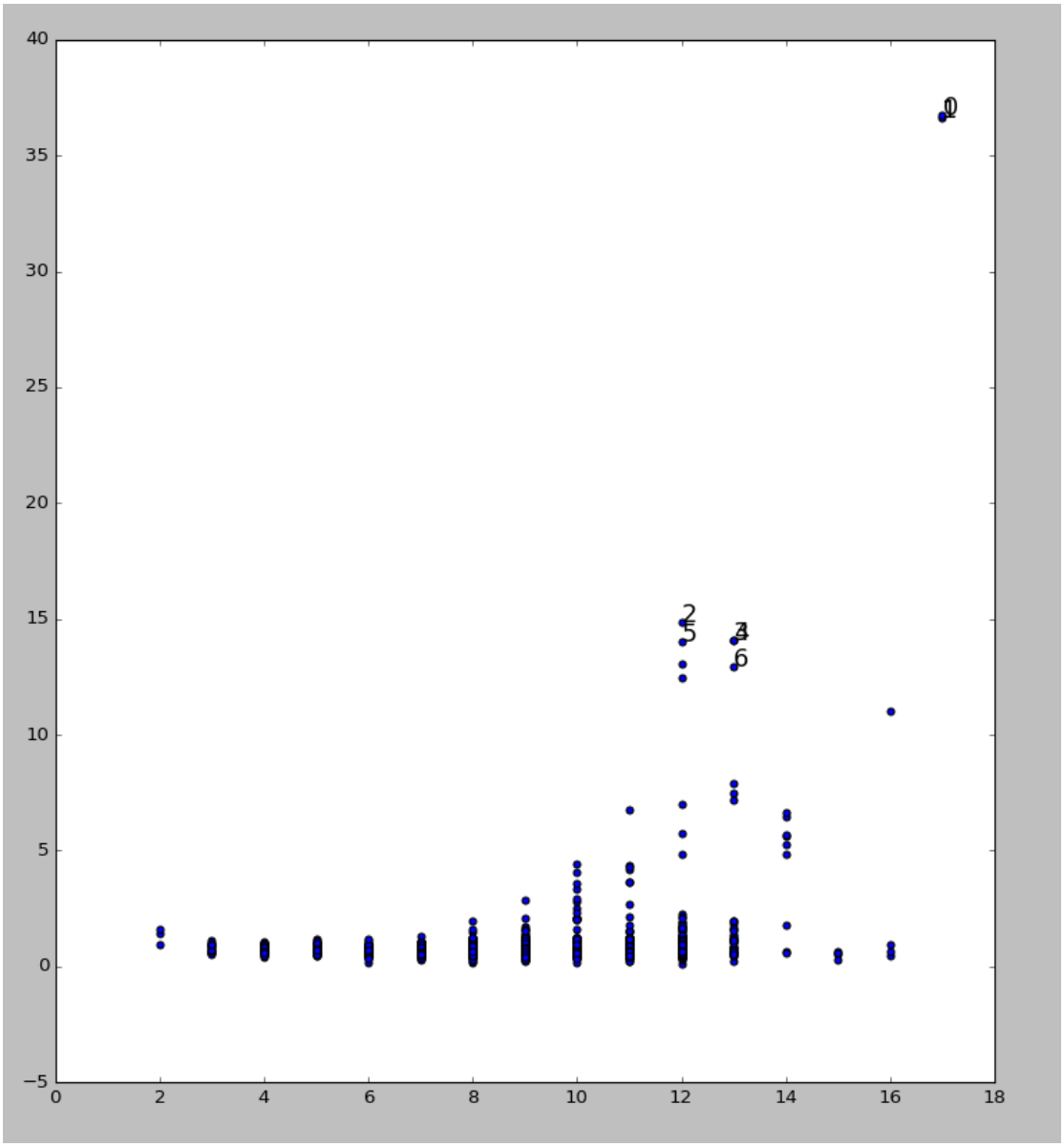
根据论文提供的计算公式：

$$\delta_i = \min_{j:\rho_j > \rho_i} (d_{ij})$$

即取密度大于当前数据点且距离当前数据点最近的点，但是根据该公式生成的结果可能会出现这样的问题。

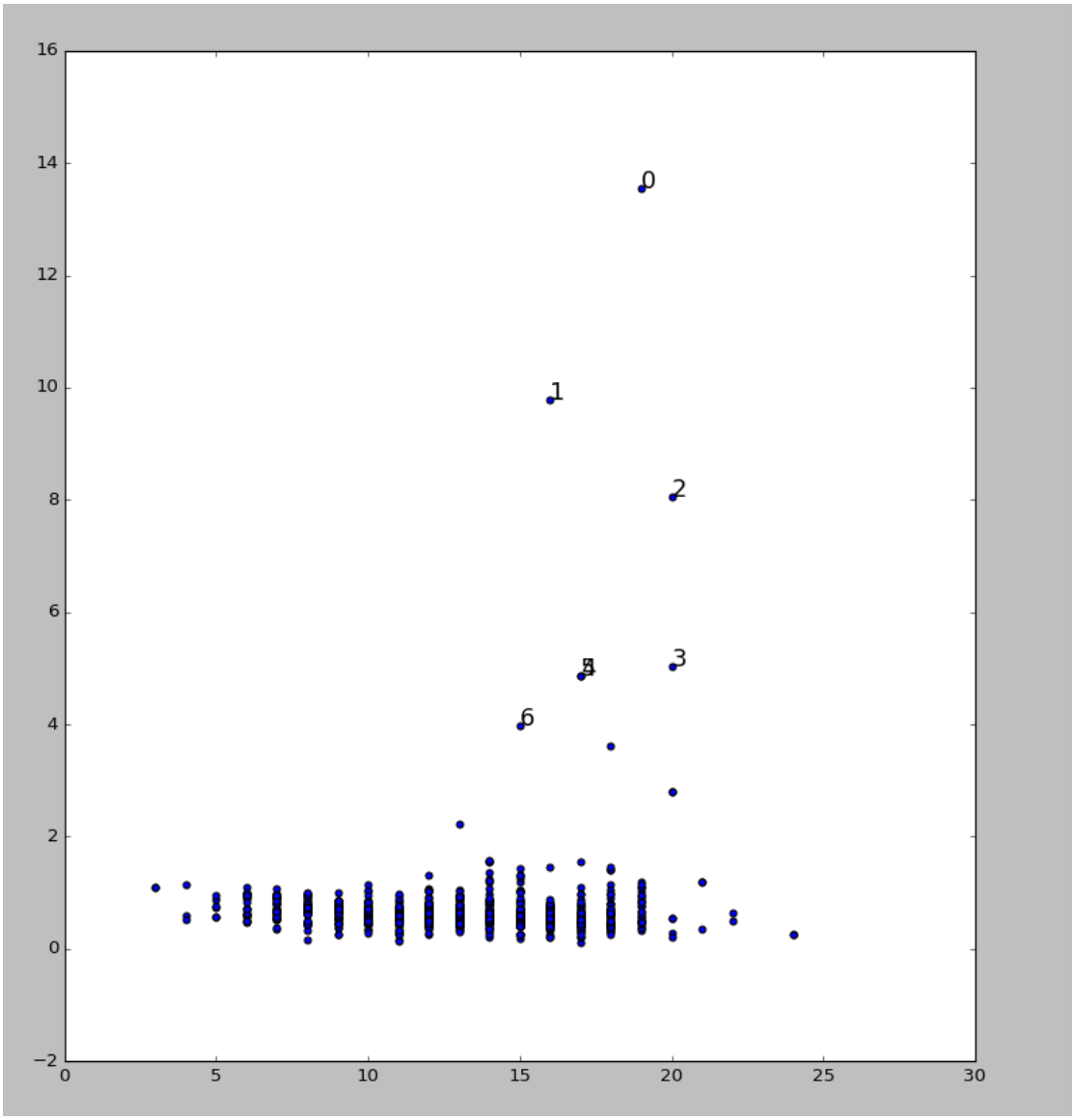


选取的数据中心 0 和 1、3 和 4、2 和 5 都是距离非常近的数据点，这也直接导致了糟糕的聚类结果。为什么会这样呢？



从 decision graph 可以看到 0 和 1、3 和 4、2 和 5 都几乎重合，意味着它们具有几乎相同的密度和距离。悲剧的原因在于 mdth 的计算公式上，因为 $\delta_i = \min_{j:\rho_j>\rho_i}(d_{ij})$ 只会考虑密度更大的数据，所以当数据点的周围出现与其密度相同的数据（而且该密度值也很大时），它会误认为当前数据点四周没有出现很大的密度值（实际上有一个离他很近的与他密度相同的点被他忽略了），这就会造成数据中心的误判。

解决方法即把公式改为“取密度***大于等于***当前数据点且距离当前数据点最近的点”。
但是这还没完，这样修改会引入新的问题：



这是修改后一次聚类的 decision grapgh。这个结果的问题在于密度最大（p=24）的数据点没有被发现，视为一个聚类中心。
这种结果产生的原因很简单，因为密度最大的点不止一个，而且相互距离很近（这种状况其实很常见），这时因为受制于 mdth 太小，所以算法不承认它是聚类中心。
当然这时可以通过调整 dc 的大小来解决，可能微调一下 dc 能让密度一样的几个点的密度错开，又会出现一个密度最大的点。但是这样的解决方案显然是不让人满意的，因为谁知道怎么调整 dc，结果无法预料。

其实，如果我们同时考虑遇到的这两个问题，会发现它们存在一定的相似性。它们都是因为几个数据点密度相同引起的。这也提醒我，当前的算法求密度的方法太粗糙，很难细致地分辨出数据点的密度差异。所以我也因此尝试寻找新的计算密度的方法。

3. local density 的计算

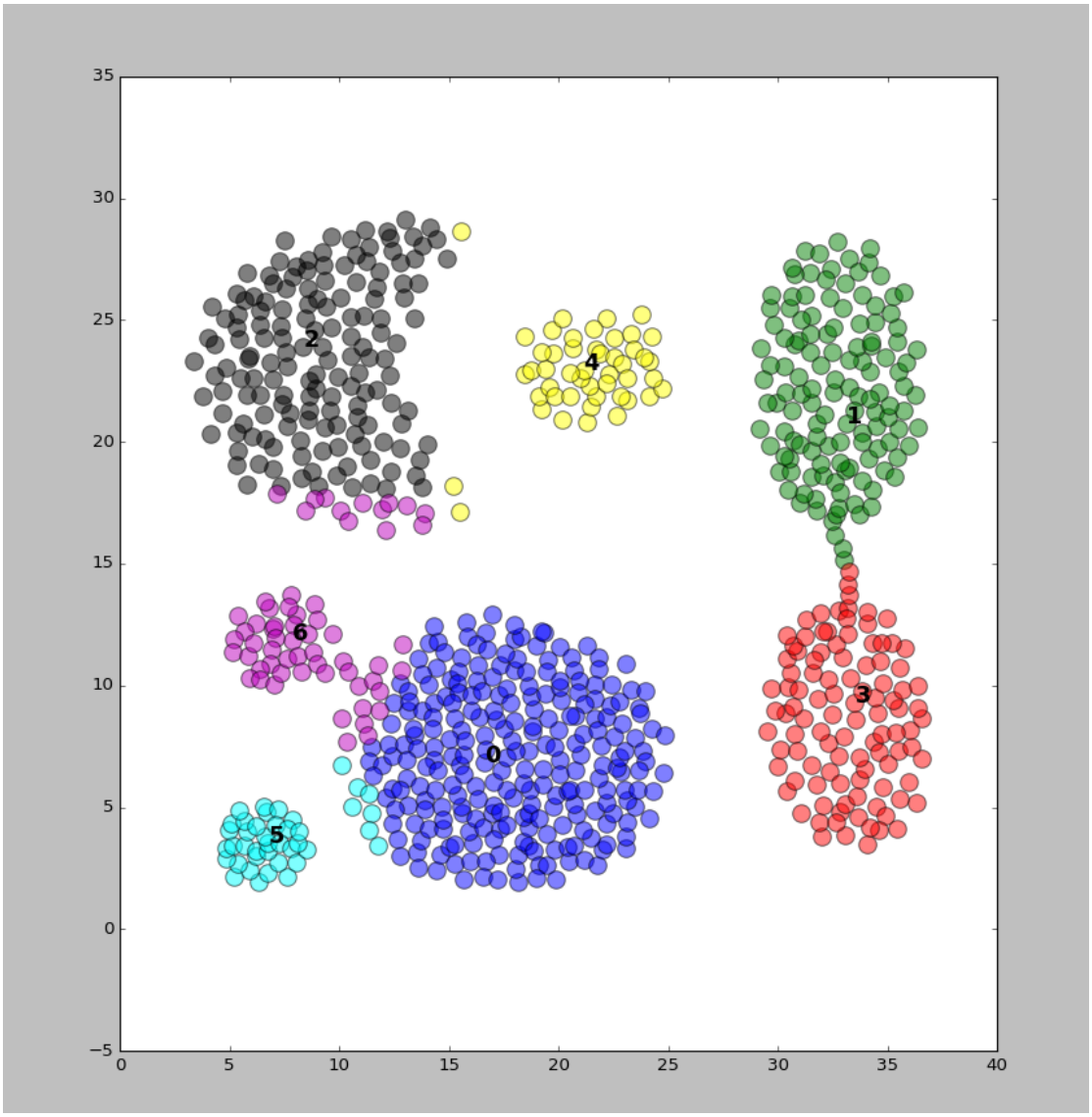
之前一直都是使用论文中 cut-off 函数计算密度

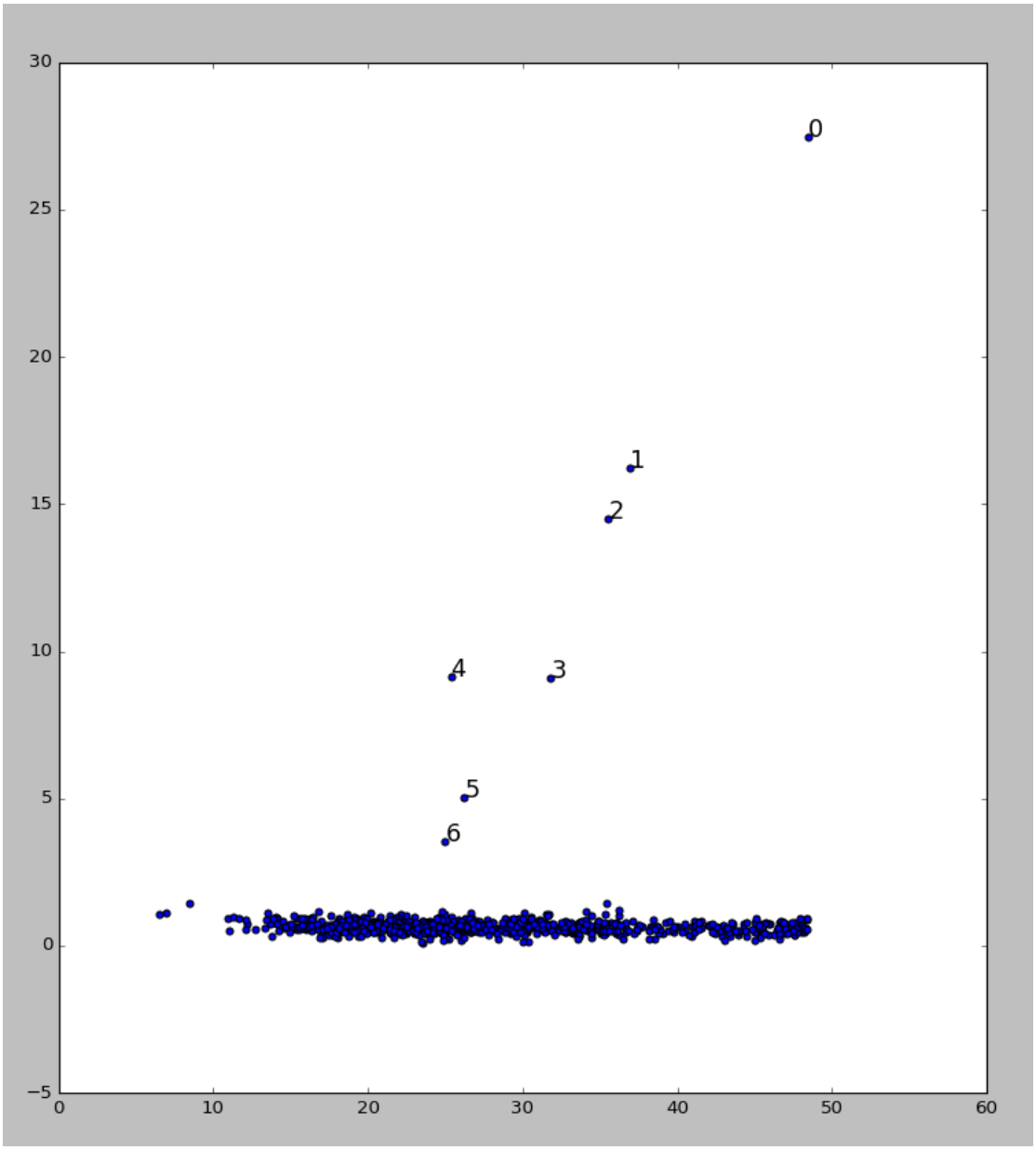
$$\rho_i = \sum_j \chi(d_{ij} - d_c) \qquad (1)$$

论文中还提到另一种高斯核函数计算密度

$$k(x, x') = e^{-\frac{||x-x'||^2}{2\sigma^2}}$$

所以我将 cut off kernal 替换为 Gaussian kernel。可以看出高斯核函数的计算公式更加复杂，这显然不如之前的截断距离函数计算简单，但是使用效果如何呢？





我发现使用高斯核函数确实提高了聚类的精度，可以看到图中能轻松的找到 6 个聚类中心。根据查到的资料——“cut-off-函数是把小于 dc 的都看成一样，而高斯核函数是根据距离衰减的。一个节点周围都是距离略小于 dc 的点（有点球状感觉 ），和一个节点周围被距离呈衰减的点围绕（类似火焰型数据），这个在 cut-off 函数算密度时候，两种节点是不区分的，但是用高斯核函数算有区别，这就是为什么火焰型数据高斯核函数相对更好点。”高斯核函数是从中心到外围有权重的距离公式，比单纯的使用截断距离包含更多的信息量，对密度的表示更精细。虽然也带来更复杂的计算，但是从结果来看，还是很有意义的。