MLlab3 实验报告

PB19030861 王湘峰

一、 实验要求

复现 Density Peak Clustering 算法，并在原论文给出的数据集上运行。将聚类结果可视化，并采用 DBI 指标评估聚类效果。具体来说需要实现以下步骤：

① 读取数据集，（如有必要）对数据进行预处理

② 实现 DPC 算法，计算数据点的ρ_i和δ_i

③ 画出决策图，选择样本中心和异常点

④ 确定分簇结果，计算评价指标，画出可视化图

二、 实验原理

首先引入密度的概念，每个点的密度ρ_i定义为与点v_i距离小于d_c的点的个数。其中d_c是人为设定的超参数。计算公式为

$$\rho_i = \sum_j \chi(d_{ij} - d_c),\ \ \chi(x) = 1\ if\ x < 0\ else\ 0$$

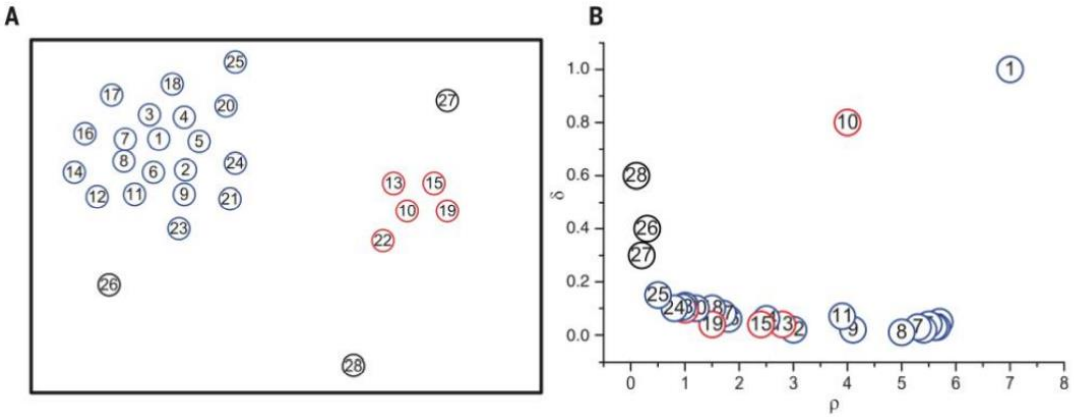
其次为每个点定义与更高密度点的距离$\delta_i = \min_{j:\rho_j > \rho_i} d_{ij}$，对于密度最高的点，它的$\delta_i = \max_j d_{ij}$

经过推理有如下结论：

① 聚类中心：同时具有较高的δ_i和ρ_i

② 离群点：具有较高的δ_i但ρ_i较低。

如果以ρ为横坐标，δ为纵坐标，则可画出决策图（如下示例）



（1,10 为聚类中心，26,27,28 为离群点）

三、 实验代码细节

① 导入需要的库函数

```
1. import pandas as pd
2. import numpy as np
3. import matplotlib.pyplot as plt
4. import plotly_express as plt1
5. from sklearn.metrics import davies_bouldin_score as dbs
6. from numba import jit
```

关于库的解释：本次实验分别导入了两个作图的库函数：matplotlib.pyplot 和 plotly_express，它们分别用来画聚类结果和决策图，plotly_express 支持查看图中点的坐标，方便确定超参数；matplotlib.pyplot 可以将聚类结果美观的呈现。由于需要计算任意两点之间的距离，这里通过距离矩阵进行存储。为了加速计算过程，使用了“黑科技”numba.jit 库来加速。

② Distance 函数计算并存储距离

```
1. @jit
2. def distance(x):
3.     m = x.shape[0]
4.     dis = np.zeros([m, m])
5.     # 距离矩阵
6.     for i in range(m - 1):
7.         for j in range(i + 1, m):
```

```
                dis[i, j] = np.sqrt(np.sum((x[i] - x[j]) ** 2))
                dis[j, i] = dis[i, j]
    return dis
```

@jit 是 numba 的修饰器，用于加速指定函数。但是 jit 不能加速类内函数，故将其定义全局函数。

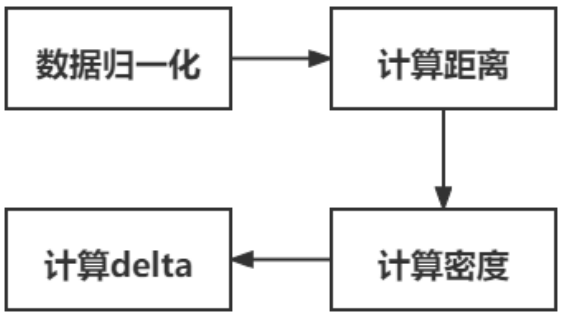
③ DPC 类的初始化

```
class DPC:

    def __init__(self, dc, thp, thd):
        self.dc = dc
        self.thp = thp
        self.thd = thd
        self.delta = []
        self.rho = []
        self.center = []
```

参数含义：dc 指计算密度时的半径，thp 指寻找聚类中心时ρ的阈值，thd 指寻找聚类中心时δ的阈值，即聚类中心需同时满足 $\rho_i >$ thp **and** $\delta_i > thd$。

④ 预处理，为每个点计算 ρ 和 δ



算法流程图

```
def process(self, x):
    # 归一化
    data = np.array((df - df.max()) / (df.max() - df.min()))
    # 计算距离
    self.dis = distance(data)
```

```
    # 计算密度
    for i in range(self.dis.shape[0]):
        rho = 0
        for j in range(self.dis.shape[0]):
            rho += 1 if self.dis[i][j] < self.dc else 0
        self.rho.append(rho)
    # 初始化 delta(全部置为 0)
    self.delta = [0] * x.shape[0]
    # 将点按照密度从大到小排列，排序后的索引记录在 index 中
    index = sorted(range(x.shape[0]), key=lambda i: self.rho[i], reverse=True)
    # 计算每个点的 delta，全局密度最大的点的 delta 置为距离最大值
    self.delta[index[0]] = self.dis[index[0]].max()
    for i in range(1, x.shape[0]):
        # index[0:i]即是密度比点 index[i]大的点的集合
        self.delta[index[i]] = self.dis[index[i]][index[0:i]].min()
    return data
```

⑤ 聚类函数 cluster（详见代码注释）

```
def cluster(self, x):
    # 保留原始数据方便后续画图
    self.x = x
    # 预处理数据
    self.process(x)
    # 初始化每个点的类别
    self.cate = [0] * x.shape[0]
    # 寻找聚类中心以及离群点，初始化类别数量 cate=1
    cate = 1
    for i in range(x.shape[0]):
        # 聚类中心
        if self.rho[i] > self.thp and self.delta[i] > self.thd:
            self.center.append(i)
            self.cate[i] = cate
            cate += 1
        #离群点
        elif self.rho[i] < self.thp and self.delta[i] > self.thd:
            self.cate[i] = -1
    # 为每个点分配类别
    index = sorted(range(x.shape[0]), key=lambda i: self.rho[i], reverse=True)
    for i in range(x.shape[0]):
        if self.cate[index[i]] == 0:
            # 每个点的类别与比其密度高的点中最近的那一个相同
            j = np.argmin(self.dis[index[i]][index[:i]])
```

```
            self.cate[index[i]] = self.cate[index[j]]
    print('共有{}个类'.format(cate - 1))
```

⑥ 作出决策图：

```
def decision(self):
    point = plt1.scatter(x=self.rho, y=self.delta)
    point.show()
```

⑦ 聚类结果可视化（聚类中心用'x'标记）

```
def show(self):
    plt.figure(figsize=(10, 6), dpi=80)
    plt.scatter(x=self.x[0], y=self.x[1], c=self.cate, marker='h')
    plt.scatter(x=self.x[0][self.center], y=self.x[1][self.center], marker='x', c='r')
    plt.show()
```

⑧ 主函数部分（以 R15 为例）

```
if __name__ == '__main__':
    df = pd.read_csv('R15.txt', header=None, sep=' ')
    # 设置超参数
    dc = 0.05
    thp = 10
    thd = 0.1
    # 聚类、可视化、量化评估
    f = DPC(dc, thp, thd)
    f.cluster(df)
    f.decision()
    f.show()
    print('DBI 得分为：', dbs(df, f.cate))
```

四、 实验困难及解决

① 数据结构设计不合理导致代码难以 debug：

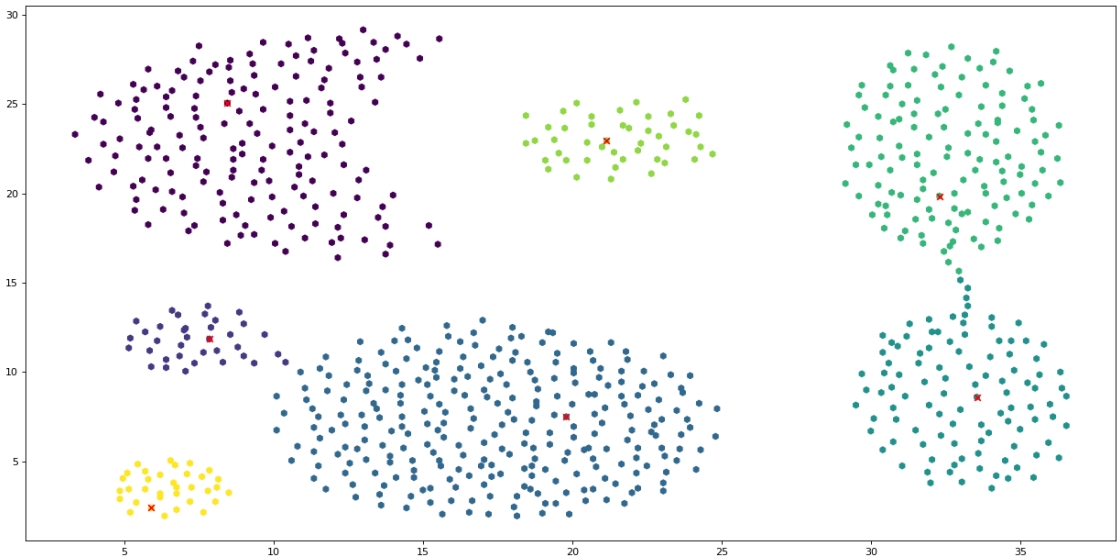
第一次实现时全部变量都放到一个矩阵中，即 column 从左到右为：索引、x 坐标、y 坐标、密度ρ、距离δ、类别 cate。这样设计导致对变量操作时会出现很多隐性的 bug 难以察觉，后来借鉴数据库的关系表的思想，将各个属性分开来保存，用哪个处理哪个，大大减少了 bug 以及精简了代码。

② Python 对于数值计算的代码执行效率低下，在计算距离矩阵时复杂度高达$O\left(\frac{n(n-1)}{2}\right)=O(n^2)$，这样给分析问题增加了大量时间成本，亟需一个更高效的方式来处理距离矩阵。

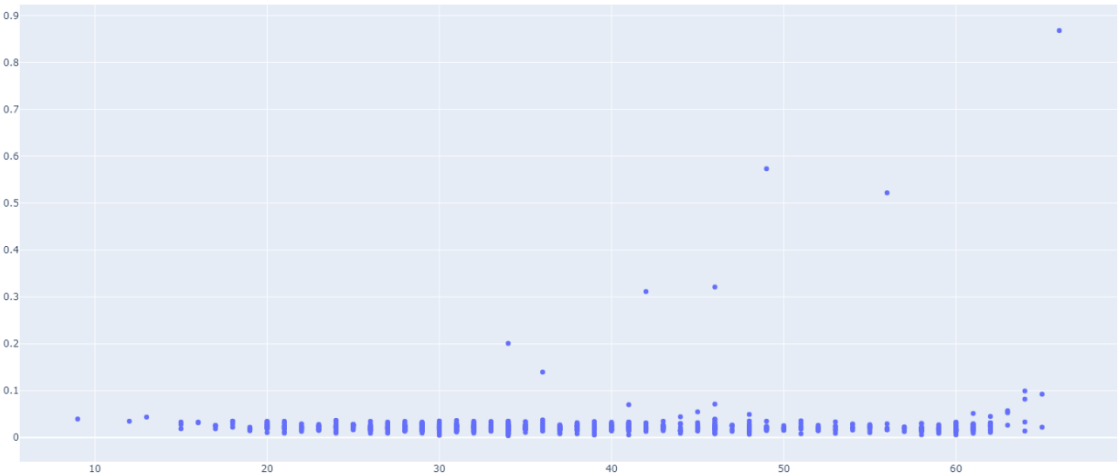
解决方法：在网上找到了可以将函数静态编译的方法，使用效率更高的编译方式来计算，大大节省了计算时间。事实上 numba 还可以调用 GPU 来实现加速，但考虑到数据的体量，杀鸡焉用牛刀，就没有开启。

五、 实验结果的展示

① Aggregation



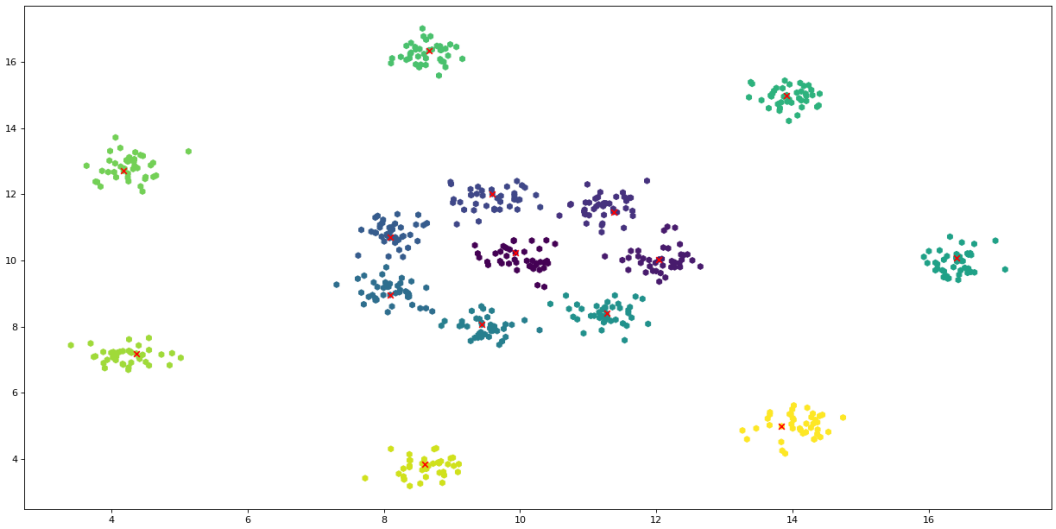
图为聚类结果，x 为聚类中心，共有 7 个类



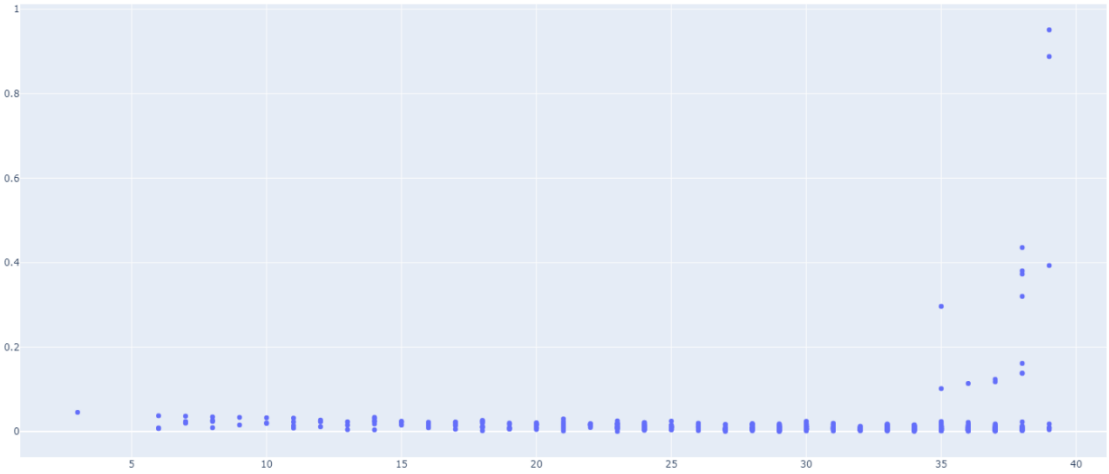
决策图，横轴为ρ，纵轴为δ

DBI： 0.5067480581049338

② **R15**



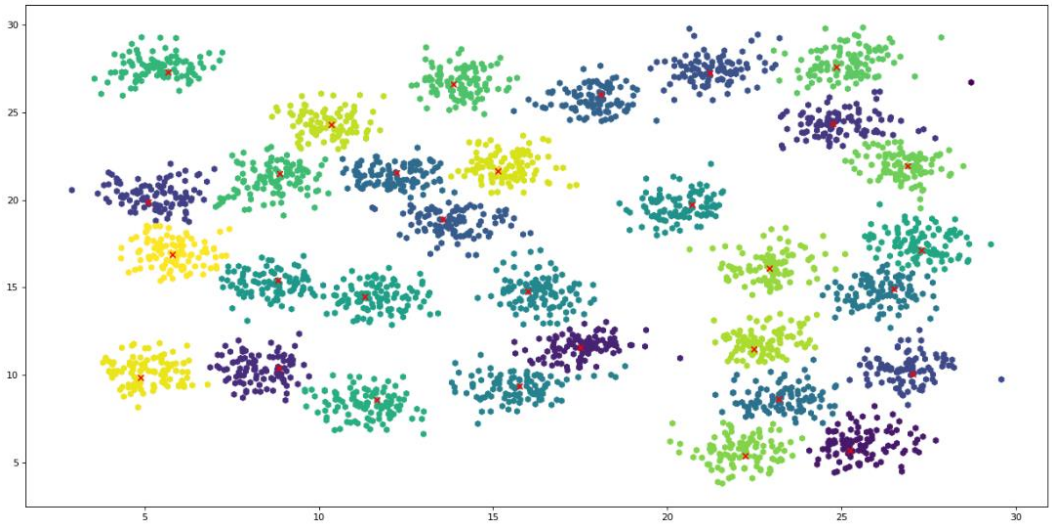
图为聚类结果，x 为聚类中心，共有 15 个类



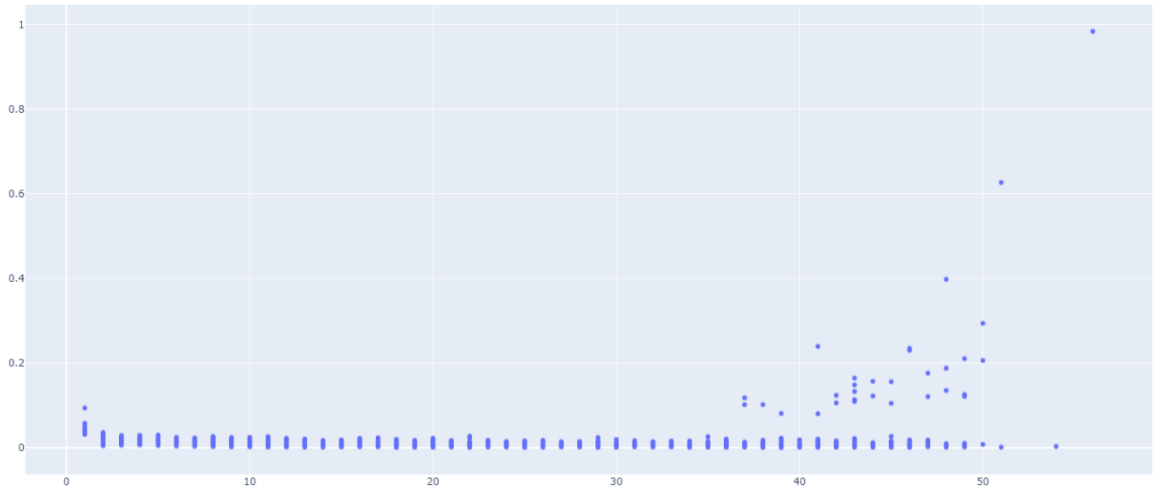
决策图，横轴为ρ，纵轴为δ

DBI： 0.31481596929442923

③ **D31**



图为聚类结果，x 为聚类中心，共有 31 个类



决策图，横轴为ρ，纵轴为δ

DBI: 0.5433114330979172