第二章实验报告

刘森栋 2019213420 2019215119

实验环境：语言：python，编译环境：python3.8，电脑系统：MacOS

第一步，数据预处理

1. 填补缺值，通过公式计算数据集 cpc 和 cpm 的特征匀值：分别为：cpc： 0.1417，cpm: 0.8345，然后通过识别两个数据集的空值部分，用各自的特征匀值替换

2. 合并数据（按时间），如下图代码所示，得到了一个初步的完整的文件，二者具有公有属性 timestamp，直接合并即可

```
df1 = pd.read_csv(r'/Users/liusendong/Desktop/original/cpc.csv', encoding="utf_8_sig")
#读取第一个文件

df2 = pd.read_csv(r'/Users/liusendong/Desktop/original/cpm.csv', encoding="utf_8_sig")

#读取第二个文件

outfile = pd.merge(df1, df2)

#文件合并 left_on左侧DataFrame中的列或索引级别用作键。right_on 右侧

outfile.to_csv(r'/Users/liusendong/Desktop/Concat/all.csv', index=False,encoding="utf_8_sig")

#输出文件

def MaxMinNormalization(x):
    x = (x - np.min(x)) / (np.max(x) - np.min(x))
    return x
```

3. 数据规格化（Max-Min 标准化），如下图代码所示，首先利用 Max-Min 标准化公式，写出函数，然后读取文件，用标准化后的数值替代原数值，将文件保存为 result.csv

```
def MaxMinNormalization(x):
    x = (x - np.min(x)) / (np.max(x) - np.min(x))
    return x


M_views = pd.read_csv('/Users/liusendong/Desktop/Concat/all.csv')
M_views['cpc'] = MaxMinNormalization(M_views[['cpc']])
M_views['cpm'] = MaxMinNormalization(M_views[['cpm']])
M_views.to_csv(r'/Users/liusendong/Desktop/Concat/result.csv', index=False,encoding="utf_8_sig")
```

4. 划分训练集和测试集，按大致 7 比 3 比例划分了训练集和测试集

第二步，算法实现

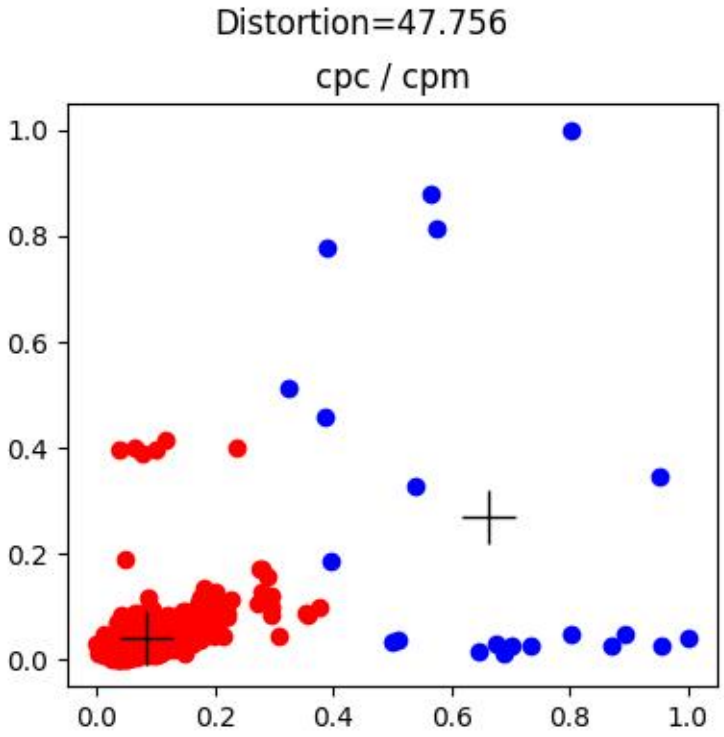
首先编写如下代码得到真异常检测图：

A.

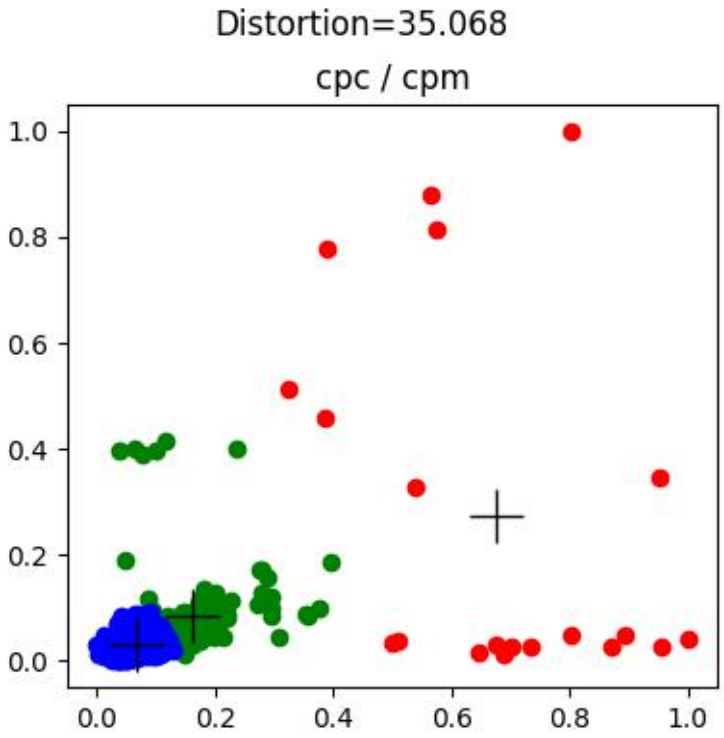
```
data=pd.read_csv('/Users/liusendong/Desktop/Concat/result_train.csv')

# 查看真异常检测二维分布图
fig, ax = plt.subplots(figsize=(10,6))
ax1 = ax.scatter(data.query("is_anomaly == True").cpc, data.query("is_anomaly == True").cpm, edgecolor = 'k', color = 'r')
ax2 = ax.scatter(data.query("is_anomaly == False").cpc, data.query("is_anomaly == False").cpm, edgecolor = 'k', color = 'b')
ax.legend([ax1, ax2], ['abnormal', 'normal'])
ax.set_xlabel('cpc')
ax.set_ylabel('cpm')
ax.set_title('Real Anomaly')
plt.show()
```

1．K-means 算法实现，首先编写好将数据集聚类的算法，见 LAB2_Kmeans 部分的代码，然后随机取 K=2，3，4，观察聚类效果。

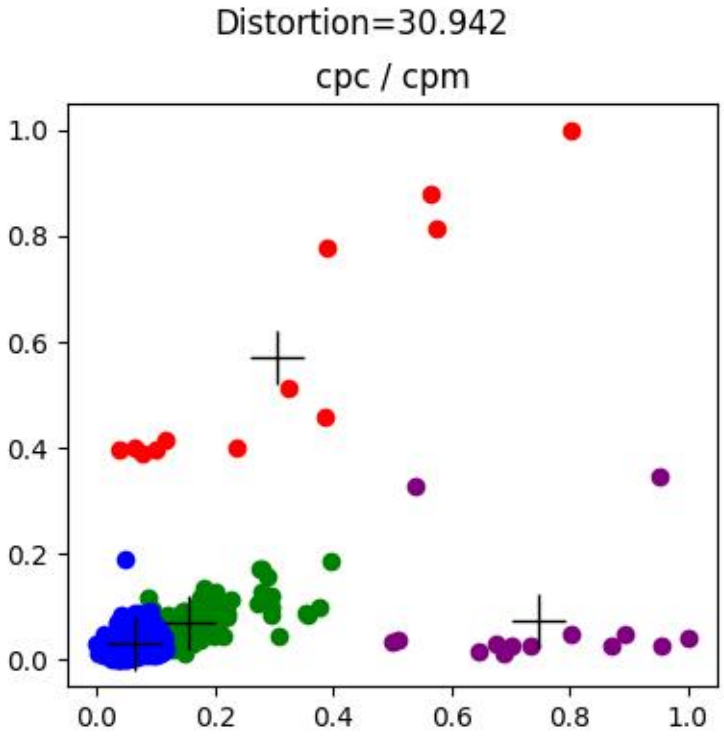
B．K=2


Distortion=47.756
cpc / cpm
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

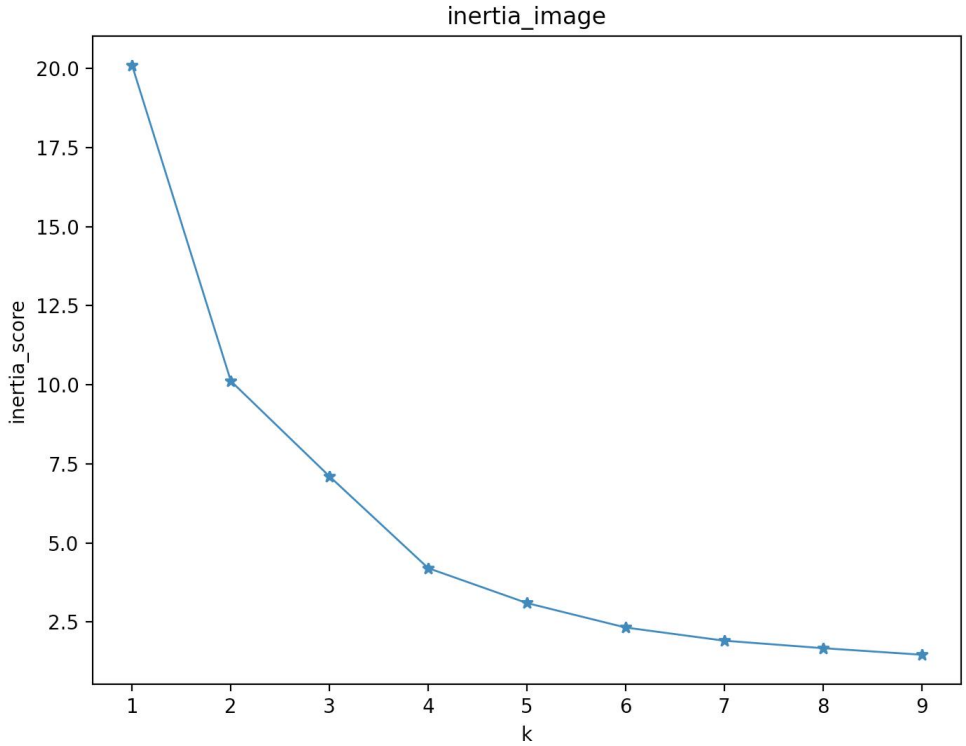
C. K=3


Distortion=35.068
cpc / cpm
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

D. K=4



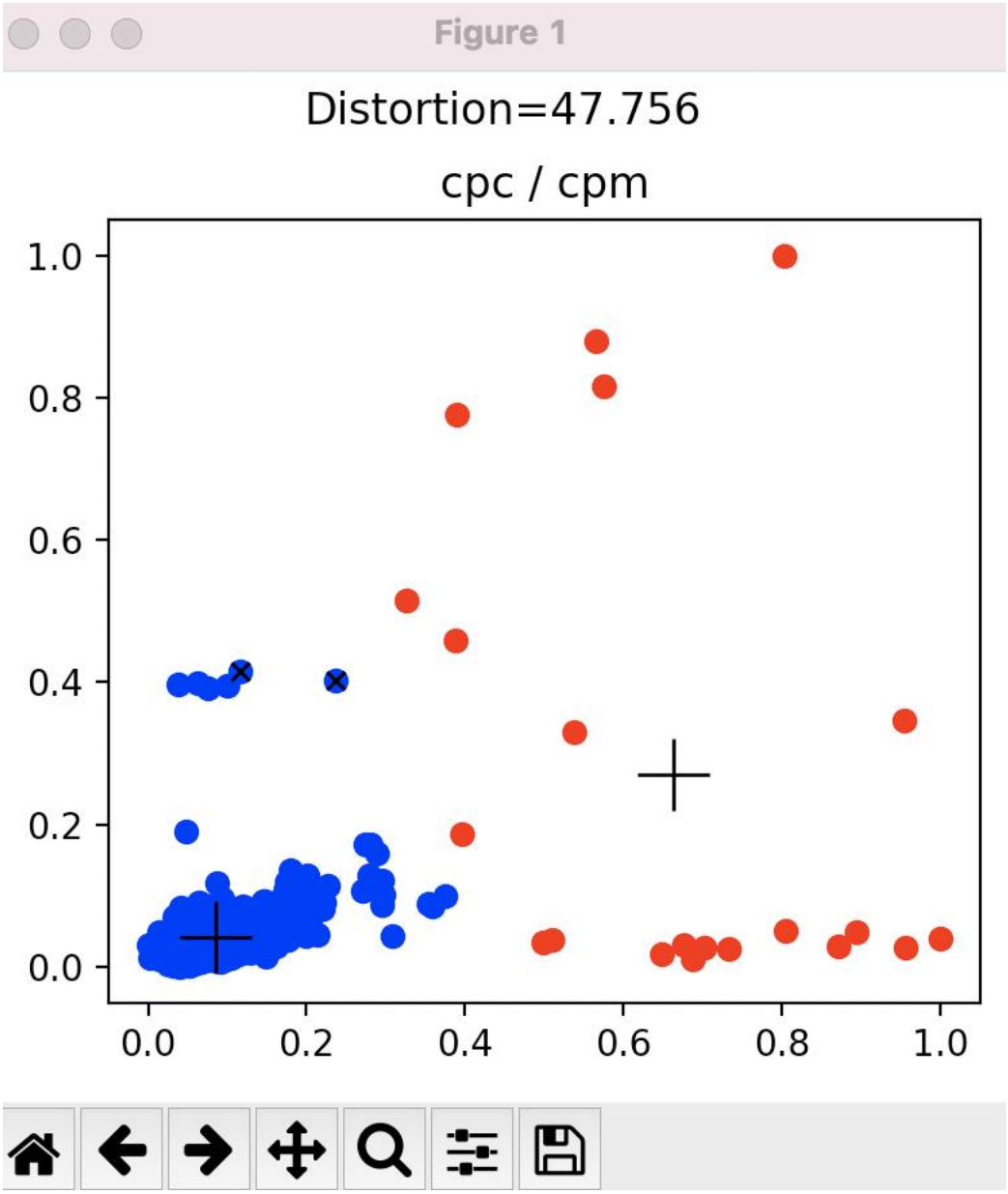
同时，可以利用肘部法则来粗略的确定 K 的取值，如下图



最终综合观察各聚类效果图和分析得到，K=2 时，聚类效果最好。

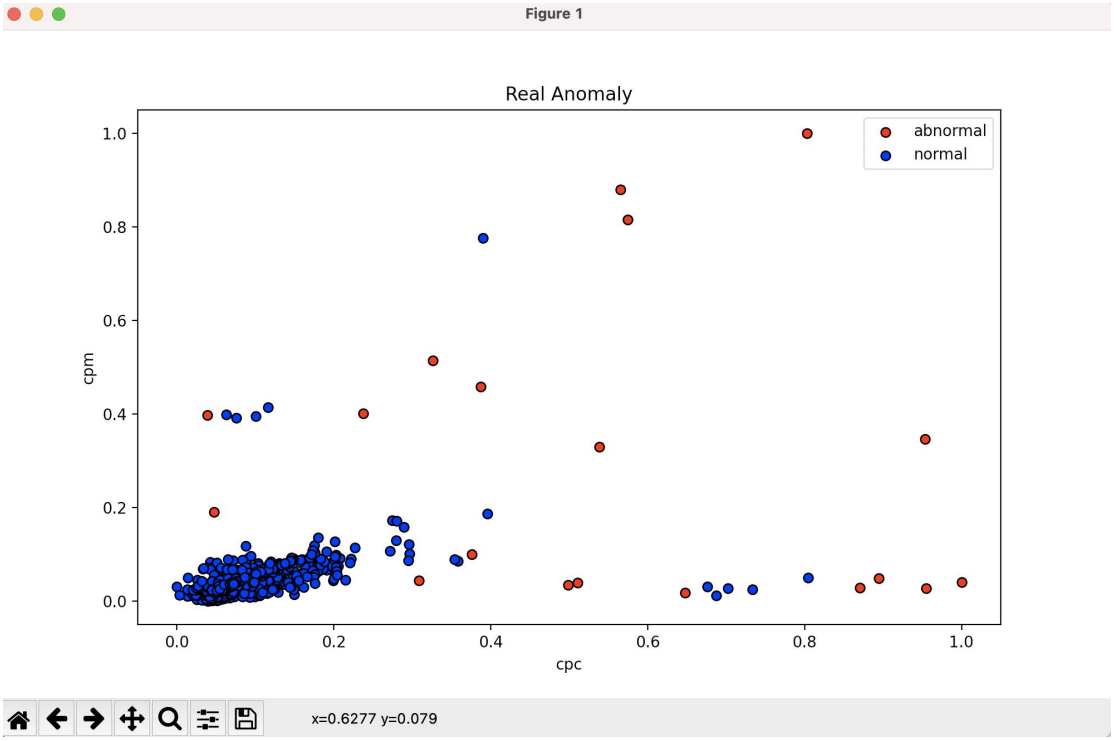
接下来进行基于 Kmeans 算法的异常检测代码的编写，核心思路是确定各聚类质心的坐标 centerxy，求出各聚类的所有点到相应质心的平均距离 avgDis, 然后利用公式求出所有点到质心相对距离 d = d (Data, centerxy）/avgDis，通过观察不同阀值时，离群点的位置和所处范围，得到最终确定的离群阀值 threshold = 8.

最后得到的结果图为



其中，蓝色部分为聚簇 1，红色部分为聚簇 2，黑色的加号为各自的质心，带黑色叉号的蓝色聚簇点以及红色聚簇点为异常点

Distortion 为畸变函数的值



与真实异常检测图进行比对，计算得到 正确率为 98.9%

结果分析：Kmeans 异常检测算法对于参数的选择比较敏感，当 K 取 3，4 时，正确率，效果很差，同时当离群阀值太小时，如小于 4，正确率也会降低，需要选择较大的离群阀值（如 8）和 K = 2，才能使得算法的准确率较高。

2. Distance-based 算法实现。

首先根据以下算法不断训练，以找到合适的 r 和 pai

```
#利用以下算法，训练并求得求得合适的r和pai
for r in rarray:
    for pai in paiarray:
        result = train_distance(r,pai,x,y)
        accuary = acctrain(result,data)
        if accuary>correct:
            correct = accuary
print(str(accuary)+' '+ str(r) +' ' + str(correct))
```

通过不断训练来得到合适的 pai 和 r 值

记录下此时的值 r：0.78，pai：0.8

然后进行基于 Distance—based 算法的异常检测代码的编写，代码如下所示

```
def train_distance(r,pai,x,y,data):
    result = [0]*len(data)
    lens =len(data)
    for i in range(0,lens):
        count = 0 #每一次循环开始重新计数
        for j in range(0,lens):
            if j-i != 0:
                #如果不是同一点，则循环计算不同点的距离
                dis = math.sqrt((x[i] - x[j]) ** 2 + (y[i] - y[j]) ** 2)
                #若距离小于r，则计数加一
                if dis <= r:
                    count = count+1
        #若计数不到总样本量的pai的比例，则将该点记做异常点，否则为正常点，用bool变量表示
        if count < pai*lens:
            result[i] = 'TRUE'
        else:
            result[i] = 'FALSE'
    return result
```

遍历整个 data 数组，如果不是同一点，则循环计算不同点的距离，若距离小于 r，计数加一，若计数小于 π*data 长度，则说明有较少的值在范围内，则说明该点异常，反之则正常，将结果标记在 result 数组中

然后执行以下代码，计算出当前 r 和 pai 值情况下的正确率：99.05%

```
output = train_distance(0.78,0.8,x_cpc,y_cpm,data)
print("accuary =")
print(acctrain(output,anomaly))
```

```
(py38) liusendongdeMBP:LAB2_distance_based liusendong$ python distancebased.py
accuary =
0.9905561385099685
```

结果分析：Distance-Based 异常检测算法相对来说比较简单，通过不断训练得到效果更好的 r 和 pai 值，改进准确率，其算法原理也比较简单易懂，但执行效率较慢，时间复杂度高（O（N2）），适合数据量小的数据集训练和测试。

3. DTI 算法实现

首先编写计算经验熵和条件经验熵的函数及其相关函数

```
@classmethod
def cal_entropy(class_obj, y):
    y = np.array(y).reshape(-1)
    counts = np.array(pd.Series(y).value_counts())
    return -((counts / y.shape[0]) * np.log2(counts / y.shape[0])).sum()

@classmethod
def cal_conditional_entropy(class_obj, x, y):
    x = np.array(pd.Series(x).sort_values()).reshape(-1)
    y = np.array(y).reshape(-1)[list(pd.Series(x).argsort())]
    split = []
    entropy = []
    for i in range(x.shape[0] - 1):
        split.append(0.5 * (x[i] + x[i + 1]))
        entropy.append((i + 1) / x.shape[0] * class_obj.cal_entropy(y[:i + 1]) + (
                1 - (i + 1) / x.shape[0]) * class_obj.cal_entropy(y[i + 1:]))
    return (np.array(entropy), np.array(split))

@classmethod
def cal_entropy_gain(class_obj, x, y):
    entropy, split = class_obj.cal_conditional_entropy(x, y)
    entropy_gain = class_obj.cal_entropy(y) - entropy
    return entropy_gain.max(), split[entropy_gain.argmax()]
@classmethod
def cal_gini(class_obj, y):
    y = np.array(y).reshape(-1)
    counts = np.array(pd.Series(y).value_counts())
    return 1 - (((counts / y.shape[0]) ** 2).sum())

@classmethod
def cal_gini_gain(class_obj, x, y):
    x = np.array(pd.Series(x).sort_values()).reshape(-1)
    y = np.array(y).reshape(-1)[list(pd.Series(x).argsort())]
    split = []
    gini = []
    for i in range(x.shape[0] - 1):
        split.append(0.5 * (x[i] + x[i + 1]))
        gini.append(
            (i + 1) / x.shape[0] * class_obj.cal_gini(y[:i + 1]) + (1 - (i + 1) / x.shape[0]) * class_obj.cal_gini(
                y[i + 1:]))
    gini_gain = class_obj.cal_gini(y) - np.array(gini)
    split = np.array(split)
    return gini_gain.max(), split[gini_gain.argmax()]
```

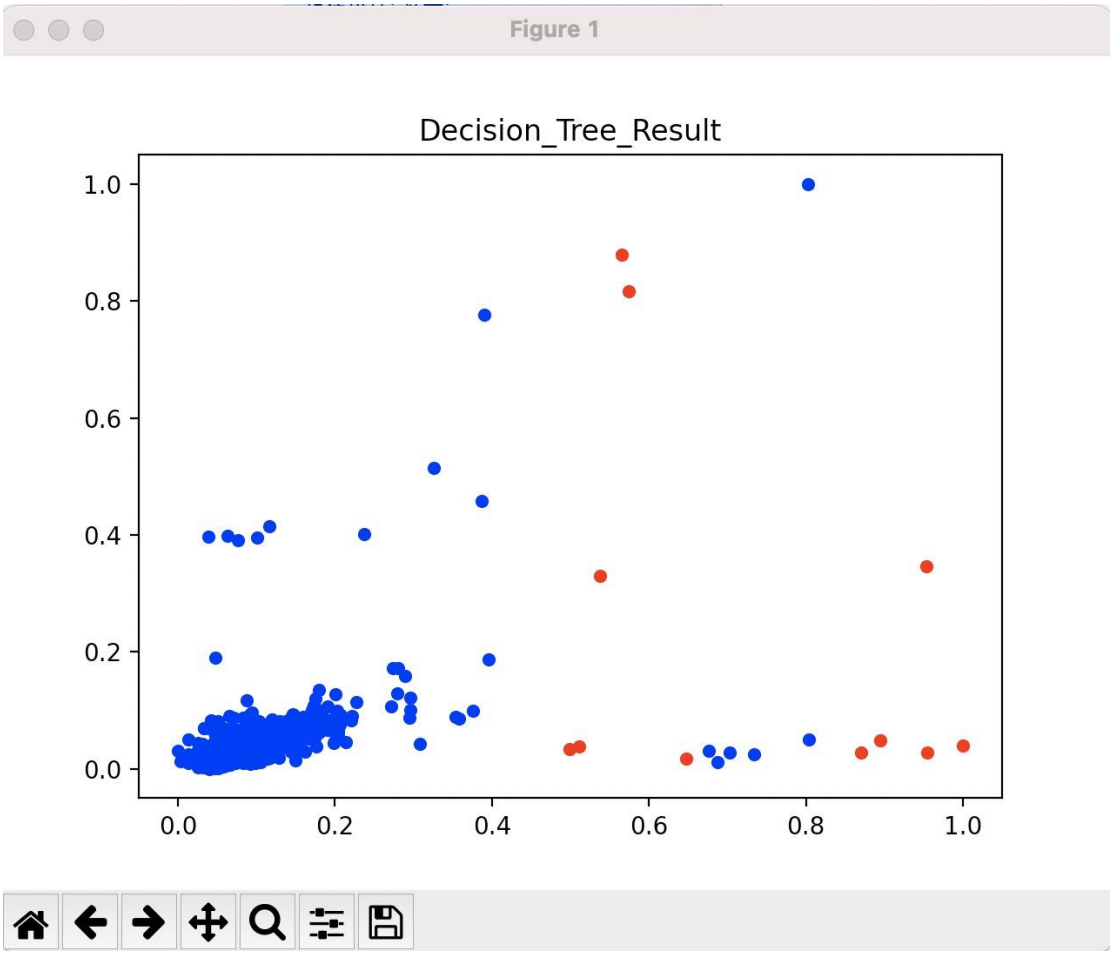
然后编写决策树结构的代码：包括初始化树，定义树的分支和节点。

```
def __init__(self, criterion="C4.5"):
    self.criterion = criterion
    self.tree_net = None

def tree(self, x, y, net):
    if pd.Series(y).value_counts().shape[0] == 1:
        net.append(y[0])
    else:
        x_entropy = []
        x_split = []
        for i in range(x.shape[1]):
            if self.criterion == "C4.5":
                entropy, split = self.cal_entropy_gain_ratio(x[:, i], y)
            else:
                entropy, split = self.cal_gini_gain(x[:, i], y)
            x_entropy.append(entropy)
            x_split.append(split)
        rank = np.array(x_entropy).argmax()
        split = x_split[rank]
        net.append(rank)
        net.append(split)
        net.append([])
        net.append([])
        x_1 = []
        x_2 = []
        for i in range(x.shape[0]):
            if x[i, rank] > split:
                x_1.append(i)
            else:
                x_2.append(i)
        x1 = x[x_1, :]
        y1 = y[x_1]
        x2 = x[x_2, :]
        y2 = y[x_2]
        return self.tree(x1, y1, net[2]), self.tree(x2, y2, net[3])

def predict_tree(self, x, net):
    x = np.array(x).reshape(-1)
    if len(net) == 1:
        return net
    else:
        if x[net[0]] >= net[1]:
            return self.predict_tree(x, net[2])
        else:
            return self.predict_tree(x, net[3])
```

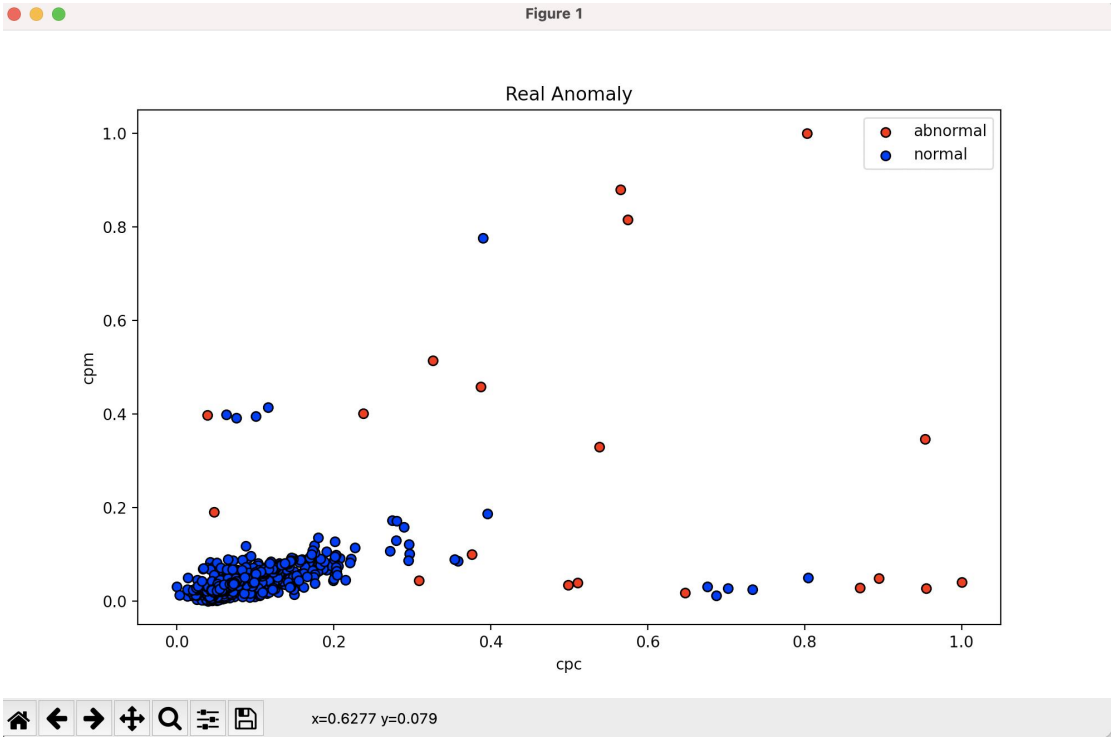
最后编写预测函数和 main 函数实现基于 DTI 的异常检测代码：DTI.py 与真实验证结果 DTI_Verify.py 进行比对，并得到预测集 predict.csv

最终得到如下结果：

A.决策树异常检测结果：



B.真异常检测图（用于验证）



通过对两图的数据同时进行统计，最终得到基于决策树的异常检测算法正确率 为 **99.06%**

比对结果如下：

结果分析：DTI 异常检测算法的训练过程比较重要，训练好分类的模型，会使得测试过程进行较快，整个算法较为复杂，但总体健壮性好，更为稳定，对中间值的缺失不敏感。

基于本次实验的数据集，综合来看 Distance-Based 算法更好，因为本次实验数据集总量不多 (1000 条左右)，Distance-Based 异常检测算法更简单易懂，方便使用，正确率也很高，很适合小规模数据集。

第三步：分析三类方法在异常检测场景中的优劣：

1.基于聚类的异常检测方法分析：

聚类和异常检测目标都是估计分布的参数，以最大化数据的总几率。

优点：

（1）无监督

（2）有坚实的统计学理论基础，当存在充分的数据和所用的检验类型的知识时，这些检验可能很是有效；

（3）测试阶段速度较快

（4）聚类密集时，类与类之间的区别明显，检测效果好

缺点：

（1）对于多纬数据，可用的选择少一些，而且对于高维数据，这些检测可能性不好

（2）如果异常数据自己成簇，将难以发现异常

（3） 对孤立点敏感

2. 基于临近度的异常检测方法分析:

这种方法比统计学方法更通常、更容易使用，由于肯定数据集的有意义的邻近性度量比肯定它的统计分布更容易

优点:

(1) 简单可用;

(2) 适应不同的类型数据类型方便，只需定义合适的便是数据间距离的方法即可

缺点:

(1) 基于邻近度的方法须要 O(m2)时间，大数据集不适用； (2) 该方法对参数的选择也是敏感的;

(2) 不能处理具备不一样密度区域的数据集，由于它使用全局阈值，不能考虑这种密度的变化。

3. 基于分类的异常检测方法分析:

复杂度:

训练阶段： 决策树会较快，基于二次最优化的算法

测试阶段： 分类技术在测试阶段会较快—分类的模型已经训练好

优点:

(1) 可以处理多分类问题

缺点:

(1) 多分类问题需要精确的多分类标签

(2) 直接将一个确定的标签赋给测试数据有时候会不恰当