《数据挖掘应用实验》课程报告

实验 1

yunwuyue lzu e-mail Student ID

Lanzhou University

Date: 2023 年 3 月 18 日

1 数据集介绍

1.1 数据集下载及分析

从 UCI dataset repository 中下载一个数据集 Breast Cancer Wisconsin (Diagnostic) Data Set[1]，形成数据集 D1，D1 以”wdbc.csv” 为文件名保存。

该数据集包含 569 个实例和 30 个特征，特征中包含一列符号型数据’Diagnosis’，作为样本的类别标签（恶性或良性），同时包含 30 列连续的数值型数据，以下每个生理指标都对应数据集中的 3 个特征（平均值、标准误差和最差值）：

- radius
- texture
- perimeter
- area
- smoothness
- compactness
- concavity
- concave points
- symmetry
- fractal dimension

表1展示了数据集 D1 的部分数据 (包含前五个连续型数值特征)，通过以下代码得到：

```
df = pd.read_csv("wdbc.csv")
print(df.iloc[0:6,0:7].to_latex(index=False))
```

1.2 添加噪声及形成数据集 D2

选择 D1 中 radius_worst 与 texture_worst 这两列数值型数据，对其中每一项数据添加上界为 40，下界为-40 的扰动，从而使其中出现离群点。

ID	Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
842302	M	17.99	10.38	122.80	1001.0	0.11840
842517	M	20.57	17.77	132.90	1326.0	0.08474
84300903	M	19.69	21.25	130.00	1203.0	0.10960
84348301	M	11.42	20.38	77.58	386.1	0.14250
84358402	M	20.29	14.34	135.10	1297.0	0.10030
843786	M	12.45	15.70	82.57	477.1	0.12780

表 1: 数据集 D1（部分）

```
df_d2 = copy.deepcopy(df[['radius_worst','texture_wost']])
noise = np.random.uniform(low=-40, high=40, size=(569, 2))
df_noisy = df_d2 + noise
```

使用 to_csv 方法将更改过的目标数据保存到指定路径，形成数据集 D2（表2）。

radius_worst	texture_wost
25.38	17.33
24.99	23.41
23.57	25.53
14.91	26.50
22.54	16.67
15.47	23.75
22.88	27.66
17.06	28.14
15.49	30.73
15.09	40.68

表 2: 数据集 D2（部分）

2 实验过程

2.1 认识数据集

分别使用 mean() 和 std() 方法输出均值和方差，结果保留五位小数。

```
for i in num_col:
    print(i,'列的均值为%.5f'%df[i].mean(),'方差为%.5f'%df[i].std(),'\n')
```

以上代码部分输出如下：

```
radius_mean 列的均值为14.12729 方差为3.52405

texture_mean 列的均值为19.28965 方差为4.30104

perimeter_mean 列的均值为91.96903 方差为24.29898
```

使用 pyplot 中的 subplot () 和 subplots () 方法来绘制多个子图，使 30 个特征的子图以 5*6 的格式输出（图1）。

图 1: D1 中数值型数据的盒图

2.2 数据标准化

2.2.1 z-score 方法

z-score 方法，也称为标准分数，是一种统计测量，用于指示一个观察值或数据点相对于一个总体或样本均值的标准差数。z 分数通过从总体或样本的均值中减去数据点，然后将结果除以总体或样本的标准差来计算 [2]。

具体地说，z-score 方法可以用以下公式表示：

$$z = \frac{x - \mu}{\sigma} \tag{1}$$

其中：

- x 是数据点或观察值
- μ 是总体或样本的均值
- σ 是总体或样本的标准差
- z 是计算得出的 z 分数

在 python 中，可以用下列代码实现 z-score 的计算, 接着使用 pd.concat 函数，将处理后的连续型数值特征的列与 ID，Diagnosis 进行拼接，将拼接后的数据集保存为 D1-zscore，用 to_LaTeX 方法输出部分 D1-zscore 的数据（表3）。

```
zscore = (df[num_col] - df[num_col].mean()) / df[num_col].std()
```

ID	Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
842302	M	1.096100	-2.071512	1.268817	0.983510	1.567087
842517	M	1.828212	-0.353322	1.684473	1.907030	-0.826235
84300903	M	1.578499	0.455786	1.565126	1.557513	0.941382
84348301	M	-0.768233	0.253509	-0.592166	-0.763792	3.280667
84358402	M	1.748758	-1.150804	1.775011	1.824624	0.280125
843786	M	-0.475956	-0.834601	-0.386808	-0.505206	2.235455

表 3: 数据集 D1_zscore（部分）

2.2.2 min-max 方法

在数据离散化中，min-max 方法是一种常用的数据标准化方法，它的目标是将原始数据映射到一个固定的区间内，通常是 [0,1] 或 [-1,1]（这里我们映射到 [0,1] 区间）。这个方法的核心思想是将原始数据中的最小值映射为 0，最大值映射为 1，其他数值按比例映射到这个区间内，这个方法可以有效地将数据映射到固定的区间内，并保持数据的相对大小关系。

具体地说，min-max 方法可以用以下公式表示：

$$x' = \frac{x - \min}{\max - \min} \tag{2}$$

- x 是原始数据
- min 是原始数据中的最小值
- max 是原始数据中的最大值
- x' 是标准化后的数据

在 python 中，可以用下列代码实现 min-max 方法, 接着使用 pd.concat 函数，将处理后的连续型数值特征的列与 ID，Diagnosis 进行拼接，将拼接后的数据集保存为 D1-minmax，用 to_LaTeX 方法输出部分 D1-minmax 的数据（表4）。

```
minmax = (df[num_col] - df[num_col].min()) / (df[num_col].max() - df[num_col].min())
```

ID	Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
842302	M	0.521037	0.022658	0.545989	0.363733	0.593753
842517	M	0.643144	0.272574	0.615783	0.501591	0.289880
84300903	M	0.601496	0.390260	0.595743	0.449417	0.514309
84348301	M	0.210090	0.360839	0.233501	0.102906	0.811321
84358402	M	0.629893	0.156578	0.630986	0.489290	0.430351
843786	M	0.258839	0.202570	0.267984	0.141506	0.678613

表 4: 数据集 D1_minmax（部分）

2.2.3 十进制小数定标法

十进制小数定标法（Decimal Scaling）将原始数据除以一个固定的基数，通常是 10 的幂次方，将数据的范围压缩到一个较小的区间内。这个基数的选择通常取决于原始数据的最大值，以确保转换后的数据能够保持足够的精度。

通过十进制小数定标法，原始数据的范围被缩小到了一个小数点后面的几位数，从而可以更方便地进行离散化处理。同时，由于十进制小数定标法是一种线性变换，因此它不会改变原始数据的排序和分布情况。

具体来说，十进制小数定标法的步骤如下：

1. 计算每一列需要缩小的基数 $scale_j$：

$$scale_j = \max \lfloor \log_{10}(|max_j|) \rfloor + 1, 0$$

2. 对于每一个元素 $x_{i,j}$，进行如下变换：

$$x'i, j = \frac{xi, j}{10^{scale_j}}$$

3. 最终得到处理后的矩阵 X'。

$$X' = [x'i, j]n \times m$$

- X 是一个输入的 $n \times m$ 的 dataFrame 结构数据
- max_j 是第 j 列的最大值
- $\lfloor \cdot \rfloor$ 表示向下取整
- $|\cdot|$ 表示取绝对值
- $x'_{i,j}$ 表示变换后的元素

在 python 中，可以用以下函数对数据进行十进制小数定标的处理, 接着使用 pd.concat 函数，将处理后的连续型数值特征的列与 ID，Diagnosis 进行拼接，将拼接后的数据集保存为 D1-float（表5）。

```
def decimal_scaling_norm(df):
    max_val = np.max(np.abs(df))
    scale = 0
    while max_val >= 1:
        scale += 1
        max_val /= 10
    return df / (10 ** scale)
```

ID	Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
842302	M	0.1799	0.1038	0.12280	0.10010	0.11840
842517	M	0.2057	0.1777	0.13290	0.13260	0.08474
84300903	M	0.1969	0.2125	0.13000	0.12030	0.10960
84348301	M	0.1142	0.2038	0.07758	0.03861	0.14250
84358402	M	0.2029	0.1434	0.13510	0.12970	0.10030
843786	M	0.1245	0.1570	0.08257	0.04771	0.12780

表 5: 数据集 D1_float（部分）

2.2.4 logistic 方法

logistic 方法通常用于对数据进行归一化处理。在 logistic 方法中，每个特征都被映射到一个 0 到 1 之间的值。具体地，对于给定的特征值 x，其 logistic 函数转换后的值 x' 为:

$$x' = \frac{1}{1 + e^{x}} \tag{3}$$

在 python 中，可以用以下函数对数据进行 logistic 的处理, 接着使用 pd.concat 函数，将处理后的连续型数值特征的列与 ID，Diagnosis 进行拼接，将拼接后的数据集保存为 D1-log（表6）。

```
def log_norm(df):
    return 1 / (1 + np.exp(-df))
```

2.3 数据离散化

2.3.1 等距离散化

等距离散化是一种简单的数据离散化方法，将数据的取值范围按照相同的间隔分成若干个区间，每个区间作为一个离散化的值。等距离散化的步骤如下：

1. 确定区间数：根据数据的分布情况和离散化的需求，选择要分成的区间数。通常情况下，区间数是由用户自行设定的，也可以根据统计学原理进行选择。

ID	Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
842302	M	1.000000	0.999969	1.0	1.0	0.529565
842517	M	1.000000	1.000000	1.0	1.0	0.521172
84300903	M	1.000000	1.000000	1.0	1.0	0.527373
84348301	M	0.999989	1.000000	1.0	1.0	0.535565
84358402	M	1.000000	0.999999	1.0	1.0	0.525054
843786	M	0.999996	1.000000	1.0	1.0	0.531907

表 6: 数据集 D1_logistic（部分）

2. 确定区间宽度：根据数据的取值范围和区间数，计算出每个区间的宽度。宽度可以通过将数据范围除以区间数得到，也可以通过其他方法确定。
3. 确定区间边界：根据区间宽度和数据的最大最小值，确定每个区间的边界。通常情况下，第一个区间的下边界设为数据的最小值，最后一个区间的上边界设为数据的最大值。其他区间的边界可以通过递推得到。
4. 进行离散化：根据确定的区间边界，将数据进行离散化。将每个数据点与各个区间的边界进行比较，确定它属于哪个区间，从而得到该数据的离散化值。

```
# 创建一个Series数据
df_radius_mean_series = copy.deepcopy(df['radius_mean'])

# 将数据划分成5个区间
df_radius_mean = df_radius_mean_series.to_frame()
df_radius_mean['bins'] = pd.cut(df_radius_mean['radius_mean'], bins=5)
```

在实验过程中，我对如何确定区间的数量（即 bin 值）进行了探究，得出了以下结论：

- 确定区间划分数需要综合考虑以下几个因素：数据分布情况、数据精度、应用场景。
- 一般来说，选择分为 5 到 10 个区间比较常见。

这里我们取 bin=5，对 D1(表1) 中 radius_mean 这一列进行等距离散化，结果如表7所示。

radius_mean	bins
17.99	(15.433, 19.658]
20.57	(19.658, 23.884]
19.69	(19.658, 23.884]
11.42	(11.207, 15.433]
20.29	(19.658, 23.884]
12.45	(11.207, 15.433]

表 7: 等距离散化后的 radius_mean 列（部分）

2.3.2 信息增益离散化

信息增益离散化是一种常用的数据离散化方法，它的主要步骤如下：

1. 根据数据的取值范围，将其分成若干个区间。一般来说，区间数可以自行设定，也可以通过某些启发式方法（如 Sturges 公式、Freedman-Diaconis 公式等）来确定。
2. 对于每个区间，计算其内部数据的信息熵。信息熵是用来衡量一个随机变量的不确定度的指标，公式为：

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{4}$$

其中，$p(x_i)$ 表示变量取值为 x_i 的概率，n 表示变量可能的取值数。

3. 对于每个区间，计算将其作为离散化后的特征对整个数据集的信息增益。信息增益是指在已知某一特征的取值情况下，对于分类问题来说，使用该特征可以带来多少信息量的提升。其计算公式为：

$$IG(x, y, split) = H(y) - [p_{left} H(y_{left}) + p_{right} H(y_{right})] \tag{5}$$

其中 $H(y)$ 是 y 的熵,p_{left} 和 p_{right} 是分裂后去往左右孩子的样本比例，而 $H(y_{left})$ 和 $H(y_{right})$ 是左右孩子的熵。

4. 选取信息增益最大的区间作为最终的离散化结果，并将该区间的取值范围作为该特征的一个取值。

```
def entropy(y):
    # 计算熵
    _, counts = np.unique(y, return_counts=True)
    p = counts / len(y)
    return -np.sum(p * np.log2(p))

def information_gain(x, y, split):
    # 计算分割点为split时的信息增益
    left_mask = x <= split
    right_mask = x > split
    left_entropy = entropy(y[left_mask])
    right_entropy = entropy(y[right_mask])
    p_left = np.sum(left_mask) / len(x)
    p_right = np.sum(right_mask) / len(x)
    return entropy(y) - (p_left * left_entropy + p_right * right_entropy)

def discretize(x, y, num_bins):
    # 将x离散化为num_bins个区间
    x_min, x_max = x.min(), x.max()
    bin_width = (x_max - x_min) / num_bins
    splits = [x_min + i * bin_width for i in range(1, num_bins)]
    gains = [information_gain(x, y, split) for split in splits]
    best_split = splits[np.argmax(gains)]
```

```
    # 将x离散化为{<=best_split, >best_split}两个值
    return np.where(x <= best_split, 0, 1)
```

2.3.3 卡方离散化

卡方离散化（Chi-Square Discretization）是一种用于将连续变量离散化的方法。它基于卡方检验的思想，将连续变量分成若干个区间，使得同一区间内的数据点的类别分布差异最小，而不同区间之间的类别分布差异最大。卡方离散化可以根据需求自由设定区间数，并且可以处理非线性的数据关系 [**liu2008discretization**]。

卡方离散化的步骤如下：

1. 对连续变量进行排序。
2. 根据需求设定区间数，并计算每个区间的宽度。
3. 将数据点按照宽度划分到对应的区间中。
4. 计算每个区间中各个类别的频数。
5. 计算每个区间中各个类别的期望频数。
6. 计算卡方值，用于评估每个区间的离散效果。
7. 若某个区间的卡方值小于设定的阈值，则将该区间与相邻区间合并，并重新计算卡方值，直到所有区间的卡方值均大于阈值。
8. 将所有区间的边界设为离散化后的分界点。

```
df_texture_mean, bins = pd.qcut(df_texture_mean_series, q=4, retbins=True, duplicates=
    'drop', labels=False)
```

这个例子中，qcut 函数将数据划分为 4 个等频区间，然后使用卡方检验来计算区间边界。retbins=True 表示返回每个区间的边界，duplicates='drop' 表示去除重复值，labels=False 表示返回每个区间的索引而不是标签。函数返回一个 Categorical 对象和一个列表，列表中的元素表示每个区间的边界。

2.3.4 CAIM 离散化

类别属性相互依赖最大化 (CAIM) 离散化的思路是将连续型变量划分成若干个区间，并将每个区间看做是一个取值。这些区间的个数需要在保证数据信息不损失的前提下尽量小。为了达到这个目的，CAIM 算法的关键是选择一个划分点，使得划分后的两个区间中，类别属性相互依赖的程度最小化。

具体来说，CAIM 算法的步骤如下:

1. 将连续型变量按照大小排序。
2. 选取每两个相邻值之间的中点作为候选划分点。
3. 对于每个候选划分点，将数据集划分为两个区间，并计算这两个区间中类别属性相互依赖的程度。这个程度可以使用个指标来度量，比如 Cramer's V 值。

4. 选择使得两个区间中类别属性相互依赖程度最小的划分点作为最终的划分点。

通过这个过程，CAIM 算法将连续型变量划分为若干个区间，每个区间可以看做是一个离散型取值。

```
def CAIM(df, col, num_bins):
    # 将数据按照指定列排序
    data = df.sort_values(by=col)[[col, 'Diagnosis']]
    data.reset_index(inplace=True, drop=True)
    n = len(data)

    # 生成候选划分点
    cut_points = [data[col].iloc[i] for i in range(num_bins - 1)]
    max_cramer_v = 0
    final_cut_point = None

    # 计算每个候选划分点的Cramer's V值
    for cut_point in cut_points:
        left = data[data[col] <= cut_point]
        right = data[data[col] > cut_point]
        n_left = len(left)
        n_right = len(right)

        # 计算左侧和右侧的类别属性分布
        left_dist = left['Diagnosis'].value_counts().sort_index().tolist()
        right_dist = right['Diagnosis'].value_counts().sort_index().tolist()

        # 计算左侧和右侧的总类别属性数量
        n_total_left = sum(left_dist)
        n_total_right = sum(right_dist)

        # 计算左侧和右侧的类别属性概率分布
        left_prob = [x / n_total_left for x in left_dist]
        right_prob = [x / n_total_right for x in right_dist]

        # 计算Cramer's V值
        p_total = (n_total_left + n_total_right) / n
        p_left = n_total_left / n
        p_right = n_total_right / n
        v = p_total * (
            sum([x**2 for x in left_prob]) / p_left +
            sum([x**2 for x in right_prob]) / p_right -
            (sum([x**2 for x in left_prob + right_prob]) / p_total)
        )

        # 更新最大Cramer's V值和最终的划分点
```

```
        if v > max_cramer_v:
            max_cramer_v = v
            final_cut_point = cut_point

    # 返回最终的划分点
    return final_cut_point
```

上述代码中，df 是一个包含原始数据的 pandas DataFrame，col 是需要离散化的列名，num_bins 是希望将该列分成的离散型取值数量。CAIM 函数会返回最终的划分点。

需要注意的是，这里使用了 Cramer's V 值作为衡量类别属性相互依赖程度的指标，而且代码中对 Cramer's V 值的计算方式也只是一种可能的方式。在实际应用中，需要根据具体情况选择合适的指标和计算方式。

现在，我们使用 CAIM 函数将 X 列离散化为 5 个取值，然后再统计每个取值下 target 变量的分布情况。

```
cut_point = CAIM(df, 'perimeter_mean', 5)
df['perimeter_mean_discrete'] = pd.cut(df['perimeter_mean'], [-np.inf, cut_point, np.
    inf], labels=[0, 1])
df.groupby('perimeter_mean_discrete')['Diagnosis'].value_counts(normalize=True)
```

上述代码中，我们将 X 列分为两个离散值，标签分别为 0 和 1，然后使用 groupby 方法计算每个离散值下 target 变量的分布情况。运行结果如下：可以看到，离散化后，连续型变量 X 被划分为了两个离散值，取值为 0 和 1。同时，我们还计算了每个离散值下 target 变量的分布情况，可以用于后续的建模分析。

2.3.5 形成 D1_discrete

按照上述四种离散化方法中的代码，对原数据集进行处理，利用 iloc 方法和 contact 函数，得到 D1_discretes 数据集，如表8所示

ID	Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
862261	B	(6.96, 11.207]	2	1	0
862485	B	(11.207, 15.433]	0	1	0
862548	M	(11.207, 15.433]	2	1	1
862717	M	(11.207, 15.433]	3	1	0
862722	B	(6.96, 11.207]	0	0	0

表 8: D1_discretes 数据集（部分）

此处给出进行离散化处理前的原数据集（表9）作为对照。

ID	Diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
862261	B	9.787	19.94	62.11	294.5
862485	B	11.600	12.84	74.34	412.6
862548	M	14.420	19.77	94.48	642.5
862717	M	13.610	24.98	88.05	582.7
862722	B	6.981	13.43	43.79	143.5

表 9: D1_discretes 数据集（部分）

2.4 离群点检测

2.4.1 LOF 算法

局部离群因子（LOF）算法是一种用于检测数据集中异常值的算法，它基于密度的概念，可以有效地识别那些在周围数据密度相对较低的点。以下是 LOF 算法的步骤：

1. 为数据集中的每个数据点计算它们与所有其他数据点之间的距离。
2. 对于每个数据点，确定它的 k 个最近邻数据点，这些最近邻数据点被称为该数据点的 k-邻域。k 的值可以通过交叉验证或经验法则进行选择。
3. 计算每个数据点的局部可达密度（LRD），它表示数据点周围区域的密度，是数据点到其 k-邻域中所有点距离的平均值的倒数。一个点的密度越大，其 LRD 越小。
4. 对于每个数据点，计算其 k-邻域中每个邻居点的局部可达密度，然后将它们的平均值除以该数据点的局部可达密度，得到它的局部离群因子（LOF）值。LOF 值表示数据点与其邻居的密度比较，如果 LOF 值接近 1，则该点与其邻居的密度相似，如果 LOF 值大于 1，则该点的密度相对较小，被认为是一个离群点。
5. 通过选择一个 LOF 值阈值，将数据集中的所有数据点分为离群点和非离群点。这个阈值可以根据经验或者交叉验证来选择。

LOF 算法可以帮助我们找到数据集中相对于其他点更为孤立的点，它不需要事先对数据进行假设或者参数化，因此适用于各种类型的数据集。

```
from sklearn.neighbors import NearestNeighbors

def LOF(X, k):
    """
    计算数据集X的局部离群因子(LOF)值。

    参数:
    X: 二维数组，表示数据集。
    k: int，表示k-邻域的大小。

    返回:
    lof_scores: 一维数组，表示每个数据点的LOF值。
```

```
    """

    # 计算每个数据点到其他数据点的距离
    nbrs = NearestNeighbors(n_neighbors=k+1).fit(X)
    distances, indices = nbrs.kneighbors(X)

    # 计算每个数据点的k-邻域可达距离
    reachability_dists = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        k_dist = distances[i, -1] # k-邻域距离
        reachability_dists[i] = max(k_dist, np.sum(distances[indices[i], -1]) / k)

    # 计算每个数据点的局部可达密度(LRD)
    lrd = 1.0 / (np.mean(reachability_dists[indices], axis=1) + 1e-10)

    # 计算每个数据点的LOF值
    lof_scores = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        lof_scores[i] = np.mean(lrd[indices[i]]) / lrd[i]

    return lof_scores
```

2.4.2 可视化离群点（绘制散点图）

```
# 使用LOF算法计算LOF值
lof_scores = LOF(df_noisy.values, k=5)

# 标识离群点
thresholds = [1, 1.1, 1.5]
fig, axs = plt.subplots(nrows=nrows, ncols=ncols, figsize=(20,6))
for i, threshold in enumerate(thresholds):
    lof_scores = LOF(df_noisy.values, k=5)
    outliers = lof_scores > threshold

    # 绘制散点图
    plt.subplot(1, 3, i+1)
    plt.scatter(df_noisy.loc[~outliers, 'radius_worst'], df_noisy.loc[~outliers, '
        texture_wost'], color='blue', label='normal')
    plt.scatter(df_noisy.loc[outliers, 'radius_worst'], df_noisy.loc[outliers, '
        texture_wost'], color='red', label='outlier')
    plt.legend()
    plt.title(f"Threshold={threshold}")

plt.tight_layout()
```

```
fig.savefig('scatterplot.png', dpi=300)
plt.show()
```

为了探究阈值 threshold 的合适取值，我进行了多次尝试，下图（图2）给出了 threshold 分别取 1、1.1、1.5 时的离群点检测情况的可视化散点图。

可以看到，threshold 越小，代表某点的 lof 值在较低水平就能判定其是离群点，也就是说，在其他条件不变的情况下，threshold 的值越小，能检测出越多的离群点。分析图2可知，threshold 设置为 1.1 是一个较为合理的数值。

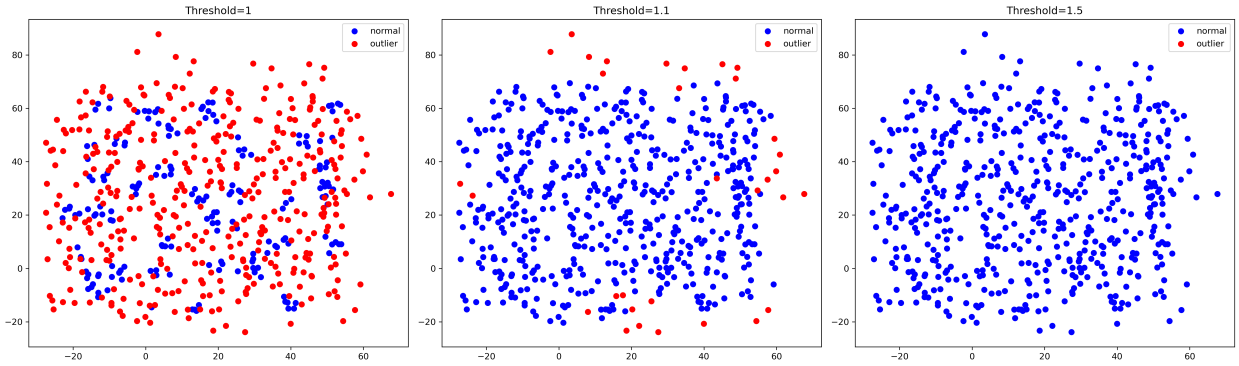

图 2: Threshold 设置为不同值时的散点图

3 结论与思考

在对原数据集进行四种不同的数据标准化操作后，我注意到使用 logistic 方法的效果不佳（见表6），原因是不少列的数据的数量级为 10^2 或者更大，在这种情况下，多数数据标准化后均为 1，可见，logistic 方法的局限性较为明显。

此外，在用 python 实现 LOF 算法时，我注意到选择适当的 k 值对于 LOF 算法的效果很重要。一般来说，k 值越大，LOF 算法越具有全局性，可以更好地发现全局离群值。但是，如果 k 值过大，LOF 算法将变得过于保守，无法发现一些局部离群值；在等距离散化划分区间时也有类似的问题：分区数量过多可能会导致过拟合，而分区数量过少可能会影响模型的准确性。因此，在为函数设置参数时需要根据具体情况进行权衡。

4 致谢

本文的 LaTeX 模板来源于 ElegantPaper。

本文中部分知识引用自课程 数据挖掘应用实验中的课件。

参考文献

[1] UCI Machine Learning Repository. *Breast Cancer Wisconsin (Diagnostic) Data Set*. 1995. URL: http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29 (visited on 11/01/1995).

[2] A. Behnood and Z. M. Nezhad. "Z-score based approach for outliers detection". In: *Communications in Statistics-Simulation and Computation* 39.5 (2010), pp. 893–910. ISSN: 0361-0918. DOI: 10.1080/03610911003719191.