程序报告

姓名：秦嘉俊　学号：3210106182

一、问题重述

1.1 背景

异常值检测（outlier detection ）是一种数据挖掘过程，用于发现数据集中的异常值并确定异常值的详细信息。当前数据容量大、数据类型多样、获取数据速度快；但是数据也比较复杂，数据的质量有待商榷；而数据容量大意味着手动标记异常值成本高、效率低下；因此能够自动检测异常值至关重要。

自动异常检测具有广泛的应用，例如信用卡欺诈检测、系统健康监测、故障检测以及传感器网络中的事件检测系统等。

1.2 任务

了解 KMeans、PCA 算法，了解算法的基本原理，并运用 KMeans 算法完成异常点检测。在实现 KMeans 算法后，尝试调整各个参数到最佳状态，或者构造更多特征，以训练出更好的模型。

二、设计思想

2.1 Kmeans 算法

K-Means 算法是最为经典的基于划分的聚簇方法，是十大经典数据挖掘算法之一。简单的说 K-Means 就是在没有任何监督信号的情况下将数据分为 K 份的一种方法。聚类算法就是无监督学习中最常见的一种，给定一组数据，需要聚类算法去挖掘数据中的隐含信息。聚类算法的应用很广：顾客行为聚类，google 新闻聚类等。

K 值是聚类结果中类别的数量。简单的说就是我们希望将数据划分的类别数。

2.1.1 要求

- 输入：n 个数据（无任何标注信息）
- 输出：k 个聚类结果
- 目的：将 n 个数据聚类到 k 个集合（也称为类簇）

2.1.2 基本概念

- 若干定义
  - n 个 m 维数据 $\{x_1, x_2, \ldots, x_n\}, x_i \in R^m (1 \leq i \leq n)$
  - 两个 m 维数据之间的欧氏距离为 $d(x_i, x_j) = \sqrt{(x_{i1} - x_{j1})^2 + \ldots + (x_{im} - x_{jm})^2}$
    $d(x_i, x_j)$ 值越小，表明 x_j 和 x_j 越相似；反之则不相似
  - 聚类集合数目 k
- 问题：如何将 n 个数据依据其相似度大小将它们分别聚类到 k 个集合，使得每个数据仅属于一个聚类集合。

2.1.3 基本思想

在数据集中根据一定策略选择 K 个点作为每个簇的初始中心，然后观察剩余的数据，将数据划分到距离这 K 个点最近的簇中，也就是说将数据划分成 K 个簇完成一次划分，但形成的新簇并不一定是最好的划分，因此生成的新簇中，重新计算每个簇的中心点，然后在重新进行划分，直到每次划分的结果保持不变。在实际应用中往往经过很多次迭代仍然达不到每次划分结果保持不变，甚至因为数据的关系，根本就达不到这个终止条件，实际应用中往往采用变通的方法设置一个最大迭代次数，当达到最大迭代次数时，终止计算。

2.1.4 具体过程

1. 初始化聚类质心
   (a) 初始化 k 个聚类质心 $c = \{c_1, c_2, \ldots, c_k\}, c_j \in R^m (1 \leq j \leq k)$
   (b) 每个聚类质心 c_j 所在的集合记为 G_j
2. 将每个待聚类数据放入唯一一个聚类集合中
   (a) 计算待聚类数据 x_i 和质心 c_j 之间的欧氏距离 $d(x_j, c_j)(1 \leq i \leq n, 1 \leq j \leq k)$
   (b) 将每个 x_i 放入与之距离最近的聚类质心所在聚类集合中，即 $argmin_{c_j \in C} d(x_i, c_j)$
3. 根据聚类结果、更新聚类质心
   根据每个聚类集合中所包含的数据，更新该聚类集合质心值，即 $c_j = \frac{1}{|G_j|} \sum_{x_i \in G_j} x_i$
4. 算法循环迭代，直到满足条件
   (a) 在新聚类质心基础上，根据欧氏距离大小，将每个待聚类数据放入唯一一个聚类集合中
   (b) 根据新的聚类结果、更新该聚类集合质心值

(c) 聚类迭代满足如下任意一个条件，则聚类停止：

- 已经达到了迭代次数上限
- 前后两次迭代中，聚类质心基本保持不变

2.1.5 K 均值聚类算法的另一个视角：最小化每个类簇的方差

方差：用来计算变量（观察值）与样本平均值之间的差异

$$argmin_G \sum_{i=1}^{k} \sum_{x \in G_i} ||x - G_i||^2 = argmin_G \sum_{i=1}^{k} |G_i| Var G_i$$

其中第 i 个类簇的方差: $var(G_i) = \frac{1}{|G_i|} \sum_{x \in G_i} ||x - G_i||^2$

- 欧氏距离与方差量纲相同
- 最小化每个类簇方差将使得最终聚类结果中每个聚类集合中所包含数据呈现出来差异性最小

2.1.6 K 均值聚类算法的不足

- 需要事先确定聚类数目，很多时候我们并不知道数据应被聚类的数目
- 需要初始化聚类质心，初始化聚类中心对聚类结果有较大的影响
- 算法是迭代执行，时间开销非常大
- 欧氏距离假设数据每个维度之间的重要性是一样的

2.2 PCA 主成分分析

主成分分析是一种特征降维方法（与线性区别分析的目的是一样的）。人类在认知过程中会主动“化繁为简”

2.2.1 基本概念

假设有 n 个数据，记为 $X = \{x_i\}(i = 1, \ldots, n)$

- 数据样本的方差 variance
  - 方差等于各个数据与样本均值之差的平方和之平均数
  - 方差描述了样本数据的波动程度

$$Var(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - u)^2$$

其中 u 是样本均值，$u = \frac{1}{n}\sum_{i=1}^{n} x_i$

- 数据样本的协方差 covariance
  衡量两个变量之间的相关度

$$cov(X,Y) = \frac{1}{n}(x_i - E(x))(y_i - E(y))$$

其中 $E(X)$ (Y) 分别是 X 和 Y 的样本均值，分别定义如下

$$E(X) = \frac{1}{n}\sum_{i=1}^{n} x_i, E(Y) = \frac{1}{n}\sum_{i=1}^{n} y_i$$

  - 对于一组两维变量（如广告投入-商品销售、天气状况-旅游出行等），可通过计算它们之间的协方差值来判断这组数据给出的两维变量是否存在关联关系：
    - 当协方差 $cov(X,Y) > 0$, 称 X 与 Y 正相关
    - 当协方差 $cov(X,Y) < 0$, 称 X 与 Y 负相关
    - 当协方差 $cov(X,Y) = 0$, 称 X 与 Y 不相关（线性意义下）

- 相关系数
  可通过皮尔逊相关系数（Pearson Correlation coefficient ）将两组变量之间的关联度规整到一定的取值范围内。皮尔逊相关系数定义如下：

$$corr(X,Y) = \frac{Cov(X,Y)}{\sqrt{Var(X)Var(Y)}} = \frac{Cov(X,Y)}{\sigma_x \sigma_y}$$

皮尔逊相关系数所具有的性质如下：

  - $|corr(X,Y) \leq 1$
  - $|corr(X,Y) = 1$ 的充要条件是存在常数 a 和 b 使得 $Y = aX + b$
  - 皮尔逊相关系数是对称的，即 $corr(X,Y) = corr(Y,X)$
  - 由此衍生出如下性质：皮尔逊相关系数刻画了变量　和　之间线性相关程度，如果 $|corr(X,Y)$ 的取值越大,则两者在线性相关的意义下相关程度越大. $|corr(X,Y) = 0$ 表示两者不存在线性相关关系（可能存在其他非线性相关的关系）。
  - 正线性相关意味着变量 X 增加的情况下，变量 Y 也随之增加；负线性相关意味着变量 X 减少的情况下，变量 Y 随之增加。

2.2.2 算法思想

- 主成分分析思想是将 n 维特征数据映射到 l 维空间（$n >> l$），去除原始数据之间的冗余性（通过去除相关性手段达到这一目的）
- 将原始数据向这些数据方差最大的方向进行投影。一旦发现了方差最大的投影方向，则继续寻找保持方差第二的方向且进行投影。
- 将每个数据从 n 维高维空间映射到 l 维低维空间，每个数据所得到最好的 k 维特征就是使得每一维上样本方差都尽可能大

2.2.3 算法描述

设有 m 条 n 维数据

- 将原始数据按列组成 n 行 m 列矩阵 X
- 将 X 的每一行进行零均值化，即减去这一行的均值
- 求出协方差矩阵 $C = \frac{1}{m}XX^\top$
- 求出协方差矩阵的特征值及对应的特征向量
- 将特征向量按对应特征值大小从上到下按行排列成矩阵，取前 k 行组成矩阵 P
- $Y = PX$ 即为降维到 k 维后的数据。

三、代码内容

3.1 Kmeans 部分

```
class KMeans():
    """
    Parameters
    ----------
    n_clusters 指定了需要聚类的个数，这个超参数需要自己调整，会影响聚类的效果
    n_init 指定计算次数，算法并不会运行一遍后就返回结果，而是运行多次后返回最好的一次结果，
        n_init即指明运行的次数
    max_iter 指定单次运行中最大的迭代次数，超过当前迭代次数即停止运行
    """

    def __init__(self, n_clusters=8, n_init=5, max_iter=50):
        self.n_clusters = n_clusters
```

```
12        self.max_iter = max_iter
13        self.n_init = n_init
14
15    def GetDistance(self, x, y):
16        """
17        计算两点之间的距离
18        """
19        return np.linalg.norm(x - y)
20
21    def GetMeans(self, x):
22        length = len(x)
23        return [sum([element[0] for element in x])/length,
24        sum([element[1] for element in x])/length, sum([element[2] for element
              in x])/length]
25
26    def fit(self, x):
27        """
28        用fit方法对数据进行聚类
29        :param x: 输入数据
30        :best_centers: 簇中心点坐标 数据类型: ndarray
31        :best_labels: 聚类标签 数据类型: ndarray
32        :return: self
33        """
34
35        max_score = float('-inf')
36        best_labels = []
37        best_centers = []
38        for init_i in range(self.n_init):
39            results = []
40            centers = []
41            for i in range(self.n_clusters): # 随机初始化聚类
42                this_center = [random.uniform(x['Dimension1'].min(),x['Dimension1
                      '].max()),random.uniform(x['Dimension2'].min(),x['Dimension2'
                      ].max()),random.uniform(x['Dimension3'].min(),x['Dimension3'
                      ].max())]
43                centers.append(this_center)
44            centers = np.array(centers)       # 便于计算距离
45            for iter_ in range(self.max_iter):
46                tmp_label = []
47                tmp_centers = []  # 二维数组，每一个元素是对应聚类所包含的数据点
48                for j in range(self.n_clusters):
49                    tmp_centers.append([])
```

```
            for i in range(len(x)): # 遍历每一个数据点
                now = np.array(x.iloc[i, 0:3])
                min_dis = float('inf')
                min_k = 0
                for j in range(self.n_clusters): # 找这个点距离最近的聚类
                    now2 = np.array(centers[j]) # 以便利用 getdistance 比较
                    if self.GetDistance(now, now2) < min_dis:
                        min_dis = self.GetDistance(now, now2)
                        min_k = j
                tmp_centers[min_k].append(now)
                tmp_label.append(min_k)
            for i in range(self.n_clusters): # 计算每一个聚类中点的中心值
                if len(tmp_centers[i])==0: # 如果聚类为空，则直接再次随机一个点
                    centers[i] = [random.uniform(x['Dimension1'].min(),x['
                        Dimension1'].max()),random.uniform(x['Dimension2'].min
                        (),x['Dimension2'].max()),random.uniform(x['Dimension3'
                        ].min(),x['Dimension3'].max())]
                else:            # 更新聚类中心的坐标
                    centers[i] = self.GetMeans(tmp_centers[i]);
            results = copy.deepcopy(tmp_label)
        now_score=silhouette_score(x, results) # 给这个聚类评分
        if now_score > max_score:
            max_score = now_score
            best_labels = copy.deepcopy(results)
            best_centers = copy.deepcopy(centers)
    self.cluster_centers_ = best_centers
    self.labels_ = best_labels
    return self
```

代码实现基本按照上文算法中的思路,对于输入的数据集 x, 我们尝试将其分为 n_clusters 簇，并每次分组我们都循环至多 max_iter 次，最后通过 n_init 次计算找到最优解。

首先我们随机初始化聚类中心，随后通过迭代尝试优化聚类中心：每次迭代我们都遍历每个数据点，计算这个点和所有聚类中心的距离，找到距离最近的那个聚类 min_k，并将其加入这个聚类。遍历完每个点后我们利用 GetMeans 重新计算每个聚类的中心值。这里需要注意的是可能存在一个聚类，但里面不含有任何点，这时我们需要再给他随机一个点以保证聚类不为空。计算结束后，将计算出的均值作为新的聚类中心，进行下一轮迭代。

如此往复，直到满足满足两个聚类迭条件之一即可退出，这样我们就完成了一次计算，我们可以利用 sklearn 里的 calinski_harabasz_score 模块来评价聚类的好坏并评分，得分的范围为 $(-1, 1)$, 得分越高聚类效果越好，方法最后返回得分最高的聚类。如此计算 n_init 次，找到得分最高的聚类即是我们算法所得到的答案。

具体细节可见代码及其注释部分。

3.2 main.py 部分

3.2.1 数据预处理

```
def preprocess_data(df):
    """
    数据处理及特征工程等
    :param df: 读取原始 csv 数据，有 timestamp、cpc、cpm 共 3 列特征
    :return: 处理后的数据，返回 pca 降维后的特征
    """
    df['timestamp'] = pd.to_datetime(df['timestamp'])
    df = df.sort_values(by='timestamp').reset_index(drop=True)
    df['cpc X cpm'] = df['cpm'] * df['cpc']
    df['cpc / cpm'] = df['cpc'] / df['cpm']
    df['hours'] = df['timestamp'].dt.hour
    df['daylight'] = ((df['hours'] >= 7) & (df['hours'] <= 22)).astype(int)
    # 请使用joblib函数加载自己训练的 scaler、pca 模型，方便在测试时系统对数据进行相同的变换
    scaler = joblib.load('./results/scaler.pkl')
    pca = joblib.load('./results/pca.pkl')
    columns = ['cpc', 'cpm', 'cpc X cpm', 'cpc / cpm']
    data = df[columns]
    data = scaler.fit_transform(data)
    data = pd.DataFrame(data, columns=columns)
    data = pca.fit_transform(data)
    data = pd.DataFrame(data,columns=['Dimension' + str(i+1) for i in range(3)
        ])
    return data
```

这里对于目前存在的 2 个特征我们引入了他们的线性、非线性组合作为新的特征。但加上 `cpc X cpm`、`cpc / cpm` 等特征之后，数据的特征数量就大于 3，无法观察数据的分布以及分多少个簇合适等；而且以上各个特征之间并不是相互独立的，存在一定程度上的相关性；故我们采用 PCA (主成分分析) 来寻找数据集的低维度表达。

在这里我们先进行数据预处理、构造特征，随后加载模型，先对各个特征进行标准化，再通过 `n_components` 指定需要降低到的维度，利用 PCA 成功降维数据。

3.2.2 异常点检测

```
def get_distance(data, kmeans, n_features):
    """
```

```
    计算距离函数
    :param data: 训练 kmeans 模型的数据
    :param kmeans: 训练好的 kmeans 模型
    :param n_features: 计算距离需要的特征的数量
    :return: 每个点距离自己簇中心的距离
    """
    distance = []
    for i in range(len(data)):
        point = np.array(data.iloc[i,:n_features])
        center = kmeans.cluster_centers_[kmeans.labels_[i]]
        distance.append(np.linalg.norm(point - center))
    distance = pd.Series(distance)
    return distance

def get_anomaly(data, kmean, ratio):
    """
    检验出样本中的异常点，并标记为 True 和 False，True 表示是异常点

    :param data: preprocess_data 函数返回值，即 pca 降维后的数据，DataFrame 类型
    :param kmean: 通过 joblib 加载的模型对象，或者训练好的 kmeans 模型
    :param ratio: 异常数据占全部数据的百分比,在 0 - 1 之间，float 类型
    :return: data 添加 is_anomaly 列，该列数据是根据阈值距离大小判断每个点是否是异常值，元素值
        为 False 和 True
    """
    num_anomaly = int(len(data) * ratio)
    new_data = deepcopy(data)
    new_data['distance'] = get_distance(new_data,kmean,n_features=len(new_data
        .columns))
    threshould = new_data['distance'].sort_values(ascending=False).reset_index
        (drop=True)[num_anomaly]
    new_data['is_anomaly'] = new_data['distance'].apply(lambda x: x >
        threshould)
    normal = new_data[new_data['is_anomaly'] == 0]
    anormal = new_data[new_data['is_anomaly'] == 1]
    return new_data
```

get_anomaly 方法中，我们设置异常点比例，在数据中添加 is_anomaly 列以反映数据是否异常。这里我们先计算出阈值距离大小，随后利用 get_distance 方法计算各个样本点到中心的距离，并借此判断样本点是否为异常点。

3.3 参数调整

值得注意的是，在 Kmeans 代码部分我们需要对 `n_clusters` `n_init` `max_iter` 赋初值，而这个初值的选取也有一定讲究。起初我就使用默认的 5、50、800, 但这样 Mo 平台的系统测试提示我“模型分数过低，请继续调整模型!”，于是我选择继续调整参数。

最后确定为 `n_clusters=5, n_init=5, max_iter=50`

四、实验结果

最后实验我们采用的参数是 `n_clusters=5, n_init=5, max_iter=50`, 我们在训练模型时得到了 0.8629563546939897 的评分，这也让我们顺利地通过了 Mo 平台的系统测试。

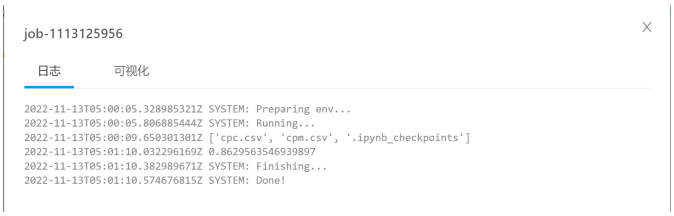

图 1: 训练模型时的得分

图 2: Mo 平台系统测试

五、总结

这次实验我们更多的是让我们自己实现 Kmeans 算法，而不是调用现成的库，这提升了我对这个算法的理解和熟悉程度，在终于实现出算法并通过测试的时候我是十分激动的。可以看到，本次实验结果基本符合预期，成功通过了测试，本地训练模型也有较高的得分。

可以发现的是,在我们训练模型时,参数并非越高越好,`n_clusters` `n_init` `max_iter` 太大，反而会导致我们的训练得分低下，无法通过系统测试。我猜测是因为当参数太大时，会存在“过拟合”的现象，这时不如减少循环次数，反而能有好的效果。

而对于算法可能的优化，我认为 `n_clusters` 可以有所改进。我最后提交的代码是将 `n_clusters` 赋为 5，但实际上这并非适合所有数据，对于簇数明显不足 5 或明显多于 5 的数据，我们的模型可能就会有很糟糕的效果。而我认为更好的方法是对于具体的数据集，我们通过一些特征或者参数，来预测这个数据集可能的簇数，随后再在这个簇数的限制下进行 Kmeans 算法，这样我们的算法能有更强的适应性。

除此之外，本次实验中我们一直在使用 Numpy 库进行相关的数值运算，这对这个库的掌握也是一个考验，很高兴我战胜了这个困难，对 Numpy 库也有了更多的了解。

最后，期待下次实验带给我新的惊喜。