第二章作业实验报告

项目准备

本数据分析作业采用基于Python 3的Jupyter Notebook进行编程，需要第三方库`pandas`，`numpy`，`matplotlab`，以及Python内置库`math`。原始工程代码为`.ipynb`格式，另有导出的`.py`源代码及`.html`格式的代码及运行结果。

数据预处理

详细的预处理程序，请查看`preprocessing.ipynb(/.py/.html)`。

填补缺失值

在原始数据中，存在一些数据确实的情况。这里采用样本均值对这些缺失的数据进行填补，如：

```
cpc_train = cpc_train.fillna(cpc_train.mean().cpc)
```

数据整合

为了检测异常，需要对数据进行cpc和cpm两项数据进行整合，如：

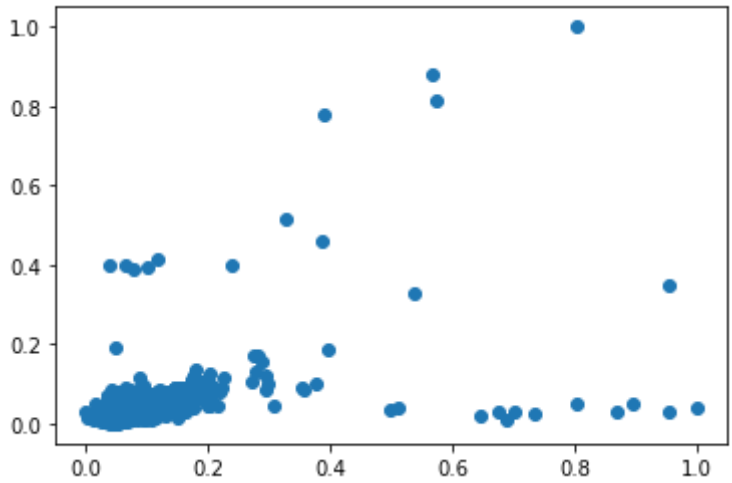
```
train = cpc_train.merge(cpm_train)
```

归一化处理

不难发现，数据中cpc的值大致在0~1的范围内，而cpm值得范围则为0~10，因此，为了更好地分析数据，需要对其进行归一化处理。这里采用最大-最小值的方法，如：

```
cpc_norm = (train["cpc"]-cpc_min)/(cpc_max-cpc_min)
```

预处理结果



K-Means算法

K-Means方法是一种基于划分的聚类分析方法，该方法将数据集分为k个聚类，并用聚类中点代表各聚类，通过不断地将数据点归至更接近的聚类，并更新中点的方式的出最终的k个聚类。因为多数点，尤其是正常点，多属于高度集中的聚类，所以在完成聚类划分的基础上，聚类中点的数量占数据集总数量小于某一临界值的聚类即可被判定为异常聚类，其中所有点被检测为异常点。

详细的K-Means算法实现程序，请查看`k_Means.ipynb(/.py/.html)`。

模型参量

输入数据为：训练或测试的归一化数据集（train_norm.csv或test_norm.csv），聚类数量k，判定临界值threshold。其中归一化的训练数据集为训练对象，k和threshold为训练目标，也是中间过程（训练）的输出量。相比于经典的K-Means算法，这里多了一个threshold，这是因为在完成聚类划分后还需要对聚类进行分类。k的预设值为{3,4,5,6,7,8,9}，threshold的预设值为{0.01,0.02,0.05,0.10）。

中间过程（训练）输出：（k，threshold）

输出结果为：对每一条数据进行异常判别标注的表（result_kMeans.csv），标有聚类和判别结果的散点图（result_kMeans.jpg）。

过程中的参数还有算法执行后的准确率precision和召回率recall，用于评估某一预设参数组（k，threshold）的效果，从而在用多组预设参数进行训练后，得出效果最佳的一组，作为训练好模型采用的参数，再利用此模型去对测试集进行检验。

算法实现

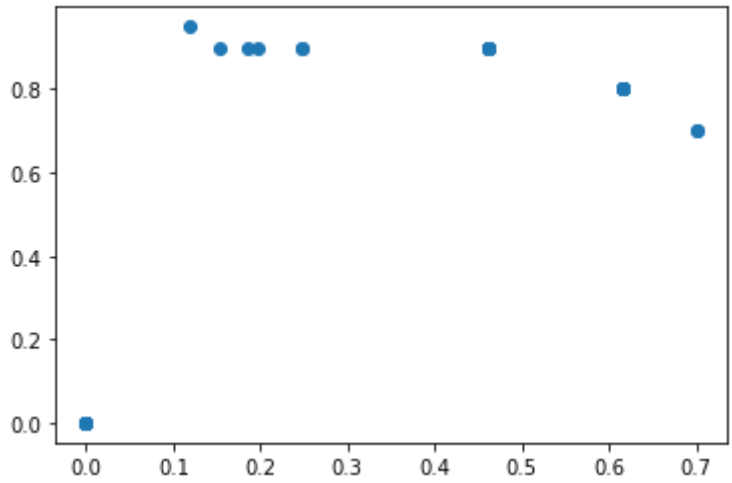
训练模型伪代码如下：

```
for each k:
    doClusterPartition()
    while(True):
        updateMeans()
        for each cluster:
            for each point:
                calculateDistanceToItsCenter()
                for each otherCluster:
                    if thisDistance > thatDistance:
                        newCluster = thatCluster
                        thisDistance = thatDistance
                if shouldMove:
                    move(thisPoint)
        if noChanges:
            break
    calculatePerformance(k,threshold)
```

测试模型与之基本一致，区别仅在于没有最外层对k的循环，以及增加了detect列表用于存储对每一个聚类的检测结果。

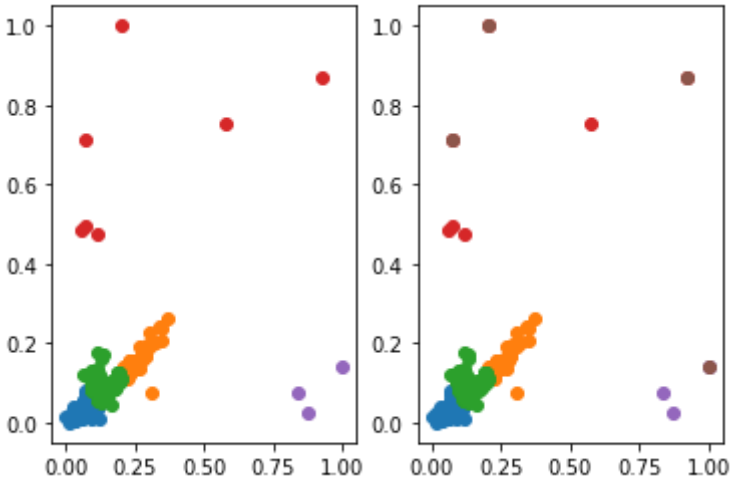
训练效果

训练后得出了28组采用不同k、threshold取值作为输入参数的模型表现，其结果如图所示。其中，横轴为precision，纵轴为recall，可见二者基本呈负相关。综合两项指标，这里采用k=5，threshold=0.02为输入参数，此时precision为0.62，recall为0.8，基本能兼顾准确率和召回率。



测试结果

得出经训练优选的输入参数k和threshold后，再次代入模型，对测试集进行聚类分析，其效果如图所示。左图中用不同颜色表示各个聚类，其中，红色和紫色的聚类被判定为异常，右图在此基础上用棕色标出了实际的异常点。



对于测试集，该模型的表现为：准确率precision=0.4，召回率recall=1.0。

具体的检测结果，可以查看`result_kMeans.csv`文件。

DTI算法

Decision Tree Induction是一种依次针对数据集不同属性进行判断，根据属性的值将各项数据分别列入不同分支，以树的形式对数据集进行拆分，最终当数据到达分支末端，也就是叶子节点时，完成分类决策。

详细的DTI算法实现程序，请查看`DTI.ipynb(/.py/.html)`。

模型准备

原则上，DTI算法需要计算数据集各属性的信息熵和信息增益，并优先选择信息增益最大的属性。不过观察数据集可以发现，数据集仅有三个属性，而从时间序列的角度看，是否异常与时间没有明显关系，且在时间范围内大体均匀分布，故“timestamp”显然不适合作为分类依据。这样就只剩下“cpc_norm”和“cpm_norm”两个属性，只能都采用，也就无需计算信息增益了。

由于属性为连续值，需要对其进行离散化处理。这里我们采用0~1/3,1/3~2/3,2/3~1这三类，划分归一化后的数据。

模型参量

输入数据为：训练或测试的归一化数据集（train_norm.csv或test_norm.csv）。

中间过程（训练）输出：各个叶子的判别情况DT_leaves

输出结果为：对每一条数据进行异常判别标注的表（result_DTI.csv），标有聚类和判别结果的散点图（result_DTI.jpg）。

有别于K-Means算法，这里不需要通过训练得出k和threshold的优选值，而作为训练结果，也就是中间过程的输出量的，是各个叶子的判别情况。（严格来说，离散化时的区间划分也可以通过训练的出最优取值，但也无非就是多几个循环的重复执行，而且考虑到数据集的大小，意义不大。）

算法实现

训练模型伪代码如下：

```
def DTI(samp): # minor parameters(bran, attr) are ignored here
    # samp - the samples(data) to be judged or splited
    if allSamples in aSameClass:
        DT_leaves[thisNode] = thisClass
    elif noMoreAttributes:
        DT_leaves[thisNode] = mode(thisClass)
    elif noSamples:
        pass
    else:
        splitSamplesIntoThreeSubSamples()
        # recursive here
        DTI(sub_sample_0)
        DTI(sub_sample_1)
        DTI(sub_sample_2)
```

模型训练

利用递归函数进行训练，得出决策树的叶子节点（真实代码）：

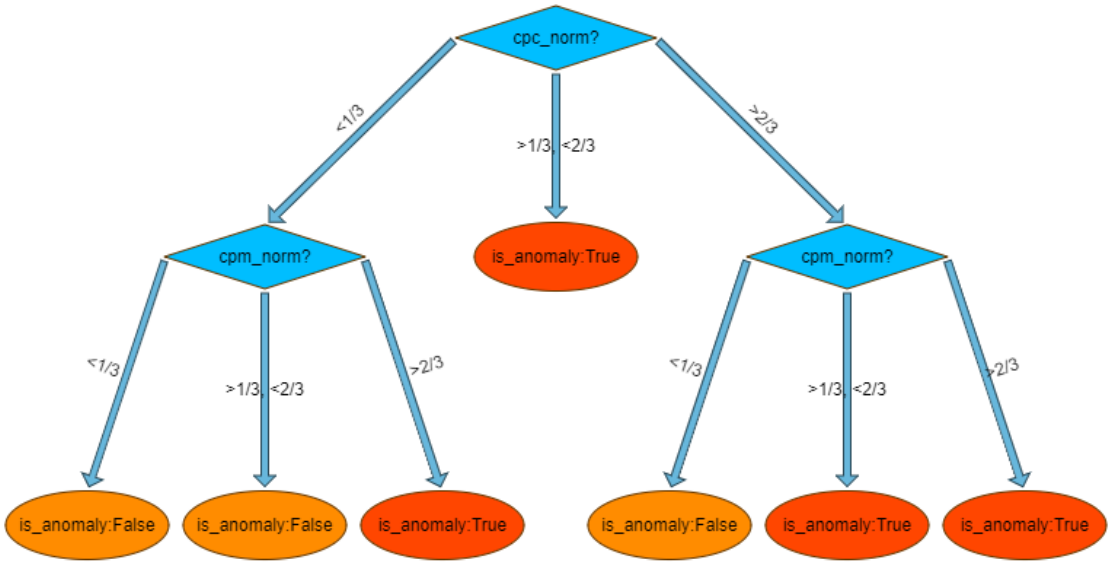
```
samples = [] #a partitions of dataset
for i in range(0,train_norm.shape[0]):
    samples.append(i) #[0-955]
attributes = ["cpc_norm", "cpm_norm"]
attr = 0 #index of attributes
bran = [-1,-1] #branches of each decision, valid locally
DT_leaves = [[False for col in range(3)] for row in range(3)] #[[fff][fff][fff]]
# train
DTI(samples,attr,bran)
```

得到训练结果`[[False, False, True], [True, True, True], [False, True, True]]`。

为了使决策树更简洁，通过trim()函数对树进行递归裁剪（真实代码）：

```
def trim(tree):
    if type(tree) == list:
        if(tree.count(tree[0]) == len(tree)):
            return tree[0]
        else:
            for i in range(len(tree)):
                tree[i] = trim(tree[i])
            return tree
    else:
        return tree
```

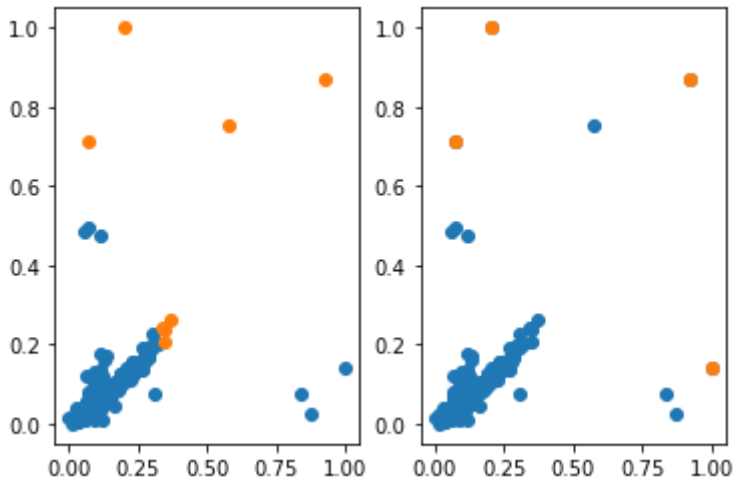
得到结果`[[False, False, True], True, [False, True, True]]`，即如下树图：



`DT_leaves`中，每一个列表元素（[]）为一个非叶子节点，非列表元素为叶子节点。

模型测试

将测试集数据按照`DT_leaves`的逻辑进行判断，得到异常检测结果如下图：



其中，左图中黄色点为检测出的异常数据，右图中黄色点为实际的异常数据。

该异常检测的准确率precision为0.375，召回率recall为0.75。

具体的检测结果，可以查看`result_DTI.csv`文件。

Distance-Based算法

Distance-Based算法通过检测一点的邻域内是否有足够的点来判断离群值，由于数据集中正常的数据点大多非常集中，可以把离群值都是为异常数据。

详细的Distance-Based算法实现程序，请查看`Distance_Based.ipynb(/.py/.html)`。

模型参量

输入数据为：训练或测试的归一化数据集（train_norm.csv或test_norm.csv），邻域半径r，判定临界值pi。其中归一化的训练数据集为训练对象，r和pi为训练目标，也是中间过程（训练）的输出量。r的预设值为{0.01,0.03,0.05,0.075,0.1}，pi的预设值为{0.001,0.003,0.005,0.01,0.02）。

中间过程（训练）输出：（r，pi）

输出结果为：对每一条数据进行异常判别标注的表（result_DistanceBased.csv），标有聚类和判别结果的散点图（result_DistanceBased.jpg）。

过程中的参数还有算法执行后的准确率precision和召回率recall，用于评估某一预设参数组（r，pi）的效果，从而在用多组预设参数进行训练后，得出效果最佳的一组，作为训练好模型采用的参数，再利用此模型去对测试集进行检验。

算法实现

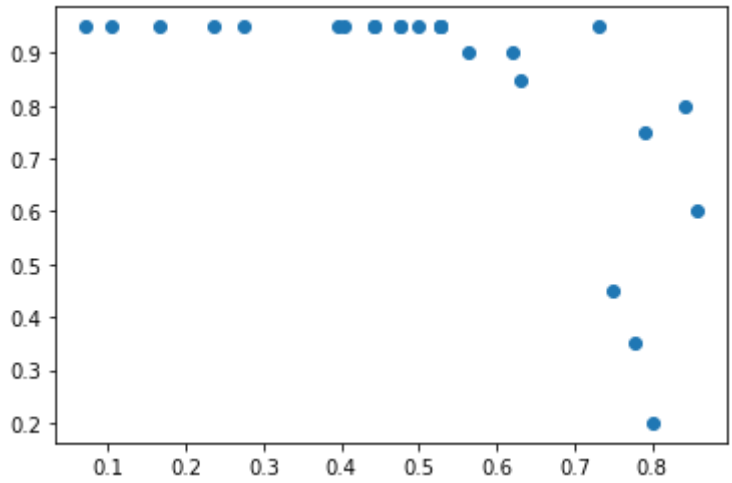
训练模型伪代码如下：

```
for each r:
    for each pi:
        candidate = all in train_norm
        for each point:
            count = 0
            for each otherPoint:
                if distance(thisPoint,thatPoint) < r:
                    count++
                    if count > pi*sizeOfDataset:
                        candidate.remove(thisPoint)
                        break
        outlier = candidate
        calculatePerformance(r,pi)
```

测试模型与之基本一致，区别仅在于没有最外层对r和pi的循环，以及增加了detect列表用于存储对每一条数据的检测结果。

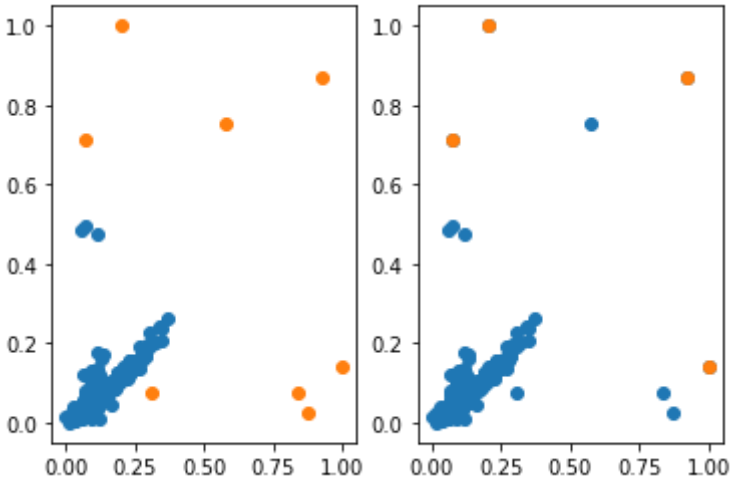
训练效果

训练后得出了25组采用不同r、pi取值作为输入参数的模型表现，其结果如图所示。其中，横轴为precision，纵轴为recall，可见二者基本呈负相关。综合两项指标，这里采用r=0.075，pi=0.003位输入参数，此时precision为0.84，recall为0.80，基本能兼顾准确率和召回率。



测试结果

得出经训练优选的输入参数r和pi后，再次代入模型，对测试集进行离群分析，其效果如图所示。左图中黄色点为检测出的异常数据，也即离群值，右图中黄色点为实际的异常数据。



对于测试集，该模型的表现为：准确率precision=0.5，召回率recall=1.0。

具体的检测结果，可以查看`result_DistanceBasedcsv`文件。

方法比较

实验表现

对于该测试集，三种方法的表现分别如下：

	precision	**recall**
K-Means	0.4	1.0
DTI	0.375	0.75
Distance Based	0.5	1.0

根据表中数据，Distance Based算法的表现最好，能识别全部的异常点，异常检测也有一半以上的准确率，而DTI表现最差。总体而言，三种方法均差强人意，基本上能实现对异常数据的检测，但"误伤"比较多（平均一半以上的误伤率）。不过也要注意到，对于训练集，K-Means的准确度也能达到0.62，而Distance Based更是有高达0.84的准确度（要不然也不会选这些模型参数），所以训练集和测试集数据本身的差异也是导致模型表现不够好的原因。

编程

从程序编写的的角度看，Distance Based算法最容易实现，对集合中的每一个点，只要分别计算其一些属性即可。相比之下，K-Means需要将一个点反复的在各个聚类之间移动，而且（由于我编程策略的原因）需要对数据集本身（由若干DataFrame组成的列表）进行操作，对编程语言的熟练程度要求较高。（后两个算法都是对0~955的数组进行操作。）DTI算法则需要用到递归函数，这是一个难点；与此同时，该算法还需要考虑如何表示树这一数据结构。

适用性

严格来说DTI并不适合用在这个应用场景中。首先因为能用来分类的属性只有两个，导致树的深度很浅，而且用cpc_norm进行分类之后，两个子节点都是用cpm_norm进行第二轮分类。这样的结果是，所谓的树事实上成了一个3*3的矩阵。此外，cpc_norm与cpm_norm的含义和分布基本一致，并不能像比较典型的DTI那样，先后针对不同类型的数据分类：如果是年轻人再看是不是学生，如果是年长者再看信用记录这样的，缺少“树”的感觉。

相比之下，对于特征属性相对较多（四、五个），属性类型差异较大（年龄、职业、性别）的，尤其是数据类型不便于用连续数据表示（取值为字符串而非浮点数）的场景，DTI就比较适用了。

而K-Means和Distance Based方法从实际含义上都比较适用该场景，无论是讨论聚类还是离群，都是在将点与数据集中的其他点作比较，这正好对应了“异常”和“正常”的概念。

Distance Based的不足则是其鲁棒性较差。当数据的特征发生变化时，检测结果可能会有较大的出入。不妨设想这样一种场景：广告运营商在制造假数据时，为了省事，所有的假数据都是一样的，这样一来在检测离群点时，这些数据的邻域内就会有大量的点，因此这些点也就不会被识别为离群点，从而逃过了异常检测。反之，如果编造的假数据比较随机，尤其是特别喜欢编造很大的数据，检测效果就会比较好。

K-Means算法中的问题在于完成聚类划分后的判断（分类）较难由程序自动实现，这里采用找出“小聚类”的方法，是基于正常点相比于异常点高度集中这一特征。但不难发现，这种分类方法从效果上与Distance Based寻找离群值是极为相似的，也就不能免于Distance Based方法的缺陷。其他的分类方法，比如用评估聚类常用的轮廓系数，也可能存在类似的问题。