西北大学

智能信息系统综合实践
实验报告

题　　目：　决策树

年　　级：　2020

专　　业：　软件工程

姓　　名：　张琬琪

一、题目

根据软木塞数据集，利用C4.5算法(不能调包)生成决策树模型。

要求：1.随机选取训练集和测试集

2.生成决策树模型，并对模型进行评估(混淆矩阵，查全率，查准率F1值)

3.使用CART算法(可调包)生成决策树模型与C4.5算法结果对比，并评价这两种算法的优缺点。

二、解题步骤（思路+代码）

【解题思路+关键代码】：

【数据集分析】：

该数据集共有 150 个数据，每个数据由编号、类别以及 10 个特征值（如 ART 、PRT 等）组成。该数据集比较均匀，是一个三分类的数据集，每一类均有 50 个数据，编号 1~50、51~100、100~150 分别对应软木塞为高质量、中等质量和低质量的数据。

【随机选取训练集和测试集】：

利用 xlrd 模块获取特征和数据标签，并对数据排版进行处理

以 3:7 的比例，随机划分测试集与训练集

①C4.5 决策树算法处理与划分数据代码，见图 1、图 2：

```
wb = xlrd.open_workbook("D:\科目学习类\pythonhomeworkCODE\exercise3\DATASETS_Cork Stoppers_Cork Stoppers.xls")
sheet = wb.sheet_by_index(1) # 获取第二个表格
# 获取第一行的数据
labels = sheet.row_values(0)  # 返回给定的行数的单元格数据进行切片
del labels[0]
del labels[0]
# 获取第二列的数据
label_num = sheet.col_values(1)
Dataset = [[], []]
for i in range(2, sheet.nrows):
    Dataset.append([])
    Dataset[i] = sheet.row_values(i)
    del Dataset[i][0]

    Dataset[i].append(Dataset[i][0])  # 将第一个元素移动到最后
    del Dataset[i][0]
del Dataset[0]
del Dataset[0]
labels.append(labels[0])
del labels[0]

test_size = 0.3
train_dataSet, test_dataSet = getDataSet(Dataset, test_size)  # 获取随机数据集
```

图 1

```
def getDataSet(dataSet, test_size):
    random.shuffle(dataSet)
    train_dataset = dataSet[:int(len(dataSet) * (1 - test_size))]
    test_dataset = dataSet[int(len(dataSet) * (1 - test_size)):]

    return train_dataset, test_dataset
```

图 2

②CART 算法处理与划分数据代码，见图 3：

```
wb = xlrd.open_workbook("D:\科目学习类\pythonhomeworkCODE\exercise3\DATASETS_Cork Stoppers_Cork Stoppers.xls")
sheet = wb.sheet_by_index(1)
labels = sheet.row_values(0)
del labels[0]
del labels[0]
label_num = sheet.col_values(1)
del label_num[0]
del label_num[0]

Dataset = [[], []]
for i in range(2, sheet.nrows):
    Dataset.append([])
    Dataset[i] = sheet.row_values(i)

    del Dataset[i][0]
    del Dataset[i][0]

del Dataset[0]
del Dataset[0]

feature_train, feature_test, target_train, target_test = train_test_split(Dataset, label_num, test_size=0.3,random_state=42)

df = pd.DataFrame(feature_train, columns=labels)
```

图 3

可以看到与前者不同在于：我们将数据标签和特征分开进行了划分，这是因为两者处理具体处理方法不同

【C4.5 决策树算法实现】：

整体思路：

(1) 属性数据处理--重点：通过连续值处理的方法，选出最佳分割点

(2) 对数据进行划分，计算各个属性的信息增益率(C4.5)-本次实验通过先计算信息增益后计算信息增益率

(3) 选择较大的信息增益率对应的划分点构建决策树

(4) 使用测试集对构建的决策树进行测试，根据预测结果，计算出各个标签的查准率、查全率、F1 的值

生成决策树模型所设定的部分功能函数：

C4.5 对连续值进行离散化的处理：

```
def calcInfoGainForSeries(dataSet, i, baseEntropy):
    # 记录最大的信息增益
    maxInfoGain = 0.0
    # 最好的划分点
    bestMid = -1
    # 得到数据集中所有的当前特征值列表
    featList = [data[i] for data in dataSet]
    # 得到分类列表
    classList = [example[-1] for example in dataSet]
    dictList = dict(zip(featList, classList))
    # 将其从小到大排序，按照连续值的大小排列
    sortedFeatList = sorted(dictList.items(), key=operator.itemgetter(0))
    # 计算连续值有多少个
    numberForFeatList = len(sortedFeatList)
    # 计算划分点，保留4位小数
    midFeatList = [round((sortedFeatList[i][0] + sortedFeatList[i + 1][0]) / 2.0, 4) for i in
                   range(numberForFeatList - 1)]
    # 计算出各个划分点信息增益
    for mid in midFeatList:
        # 将连续值划分为不大于当前划分点和大于当前划分点两部分
        eltDataSet, gtDataSet = splitDataSetForSeries(dataSet, i, mid)

        newEntropy = len(eltDataSet) / len(dataSet) * calcShannonEnt(eltDataSet) + len(gtDataSet) / len(dataSet) * calcShannonEnt(gtDataSet)
        # 计算出信息增益
        infoGain = baseEntropy - newEntropy

        if infoGain > maxInfoGain:
            bestMid = mid
            maxInfoGain = infoGain
    return maxInfoGain, bestMid
```

给定训练集D和连续属性a，假定a在D上出现了 n 个不同的取值，先把这些值从小到大排序，记为$\{a^1, a^2, \ldots, a^n\}$. 基于划分点 t 可将D分为子集D_t^-和D_t^+，其中D_t^-是包含那些在属性a上取值不大于 t 的样本，D_t^+则是包含那些在属性a上取值大于 t 的样本。显然，对相邻的属性取值a^i与a^{i+1}来说，t 在区间$[a^i, a^{i+1})$中取任意值所产生的划分结果相同。因此，对连续属性a，我们可考察包含$\mathrm{n}-1$个元素的候选划分点集合

$$T_a = \left\{ \frac{a^i + a^{i+1}}{2} \mid 1 \le i \le n-1 \right\}$$

即把区间$[a^i, a^{i+1})$的中位点$\frac{a^i + a^{i+1}}{2}$作为候选划分点。然后，我们就可以像前面处理离散属性值那样来考虑这些划分点，选择最优的划分点进行样本集合的划分，使用的公式如下：

$$Gain(D, a) = \max_{t \in T_a} Gain(D, a, t) = \max_{t \in T_a} \left(Ent(D) - \sum_{\lambda \in \{-,+\}} \frac{|D_t^\lambda|}{|D|} Ent(D_t^\lambda) \right)$$

其中$Gain(D, a, t)$是样本集D基于划分点 t 二分后的信息增益。划分的时候，选择使$Gain(D, a, t)$最大的划分点。

计算信息增益与信息增益率：

```
def calcInfoGain(dataSet, featList, i, baseEntropy):
    # 将当前特征唯一化，也就是说当前特征值中共有多少种
    uniqueVals = set(featList)
    # 新的熵，代表当前特征值的熵
    newEntropy = 0.0
    # 遍历现在有的特征的可能性
    for value in uniqueVals:
        # 在全部数据集的当前特征位置上，找到该特征值等于当前值的集合
        subDataSet = splitDataSet(dataSet=dataSet, axis=i, value=value)
        # 计算出权重
        prob = len(subDataSet) / float(len(dataSet))
        # 计算出当前特征值的熵
        newEntropy += prob * calcShannonEnt(subDataSet)

    # 计算出“信息增益”
    infoGain = baseEntropy - newEntropy

    return infoGain
```

```
def chooseBestFeatureToSplit(dataSet, labels):

    # 得到数据的特征值总数
    numFeatures = len(dataSet[0]) - 1
    # 计算出基础信息熵
    baseEntropy = calcShannonEnt(dataSet)
    # 基础信息增益为0.0
    bestInfoGain = 0.0
    #基础信息增益率为0.0
    bestInfoGainRate = 0.0

    # 最好的特征值
    bestFeature = -1
    # 标记当前最好的特征值是不是连续值
    flagSeries = 0
    # 如果是连续值的话，用来记录连续值的划分点
    bestSeriesMid = 0.0
```

```
    # 对每个特征值进行求信息熵
    for i in range(numFeatures):
        # 得到数据集中所有的当前特征值列表
        featList = [example[i] for example in dataSet]

        if isinstance(featList[0], str):
            infoGain = calcInfoGain(dataSet, featList, i, baseEntropy)
        else:
            # print('当前划分属性为：' + str(labels[i]))
            infoGain, bestMid = calcInfoGainForSeries(dataSet, i, baseEntropy)

        prob = float(i + 1) / len(dataSet[0])

        infoGainRate = infoGain / -(prob * log(prob, 2) + (1 - prob) * log(1 - prob, 2))

        # 如果当前的信息增益率比原来的大
        if infoGainRate > bestInfoGainRate:
            # 最好的信息增益
            bestInfoGainRate = infoGainRate
            # 新的最好的用来划分的特征值
            bestFeature = i

            flagSeries = 0
            if not isinstance(dataSet[0][bestFeature], str):
                flagSeries = 1
                bestSeriesMid = bestMid
    if flagSeries:
        return bestFeature, bestSeriesMid
    else:
        return bestFeature

def majorityCnt(classList):
    # 用来统计标签的票数
    classCount = collections.defaultdict(int)
    # 遍历所有的标签类别
    for vote in classList:
        classCount[vote] += 1
    # 从大到小排序
    sortedClassCount = sorted(classCount.items(), key=operator.itemgetter(1), reverse=True)
    # 返回次数最多的标签
    return sortedClassCount[0][0]
```

```python
def createTree(dataSet, labels):
    # 拿到所有数据集的分类标签
    classList = [example[-1] for example in dataSet]
    # 统计第一个标签出现的次数，与总标签个数比较，如果相等则说明当前列表中全部都是一种标签，此时停止划分
    if classList.count(classList[0]) == len(classList):
        return classList[0]

    # 计算第一行有多少个数据，如果只有一个的话说明所有的特征属性都遍历完了，剩下的一个就是类别标签
    if len(dataSet[0]) == 1:
        # 返回剩下标签中出现次数较多的那个
        return majorityCnt(classList)

    # 选择最好划分特征，得到该特征的下标
    bestFeat = chooseBestFeatureToSplit(dataSet=dataSet, labels=labels)

    # 得到最好特征的名称
    bestFeatLabel = ''

    # 记录此刻是连续值还是离散值,1连续，2离散
    flagSeries = 0

    # 如果是连续值，记录连续值的划分点
    midSeries = 0.0

    # 如果是元组的话，说明此时是连续值
    if isinstance(bestFeat, tuple):
        # 重新修改分叉点信息
        bestFeatLabel = str(labels[bestFeat[0]]) + '=' + str(bestFeat[1])
        # 得到当前的划分点
        midSeries = bestFeat[1]
        # 得到下标值
        bestFeat = bestFeat[0]
        # 连续值标志
        flagSeries = 1
    else:
        # 得到分叉点信息
        bestFeatLabel = labels[bestFeat]
        # 离散值标志
        flagSeries = 0

    myTree = {bestFeatLabel: {}}
    featValues = [example[bestFeat] for example in dataSet]

    # 连续值处理
    if flagSeries:
        # 将连续值划分为不大于当前划分点和大于当前划分点两部分
        eltDataSet, gtDataSet = splitDataSetForSeries(dataSet, bestFeat, midSeries)
        # 得到剩下的特征标签
        subLabels = labels[:]
        # 递归处理小于划分点的子树
        subTree = createTree(eltDataSet, subLabels)
        myTree[bestFeatLabel]['小于'] = subTree
        # 递归处理大于当前划分点的子树
        subTree = createTree(gtDataSet, subLabels)
        myTree[bestFeatLabel]['大于'] = subTree
```

```
        return myTree

    # 离散值处理
    else:

        # 将本次划分的特征值从列表中删除掉
        del (labels[bestFeat])
        # 唯一化，去掉重复的特征值
        uniqueVals = set(featValues)
        # 遍历所有的特征值
        for value in uniqueVals:
            # 得到剩下的特征标签
            subLabels = labels[:]
            # 递归调用，将数据集中该特征等于当前特征值的所有数据划分到当前节点下，递归调用时需要先将当前的特征去除掉
            subTree = createTree(splitDataSet(dataSet=dataSet, axis=bestFeat, value=value), subLabels)
            # 将子树归到分叉处下
            myTree[bestFeatLabel][value] = subTree
        return myTree
```

```
# 输入三个变量（决策树，属性特征标签，测试的数据）
def classify(inputTree, featLables, testVec):
    firstStr = list(inputTree.keys())[0]
    secondDict = inputTree[firstStr]  # 树的分支，子集合Dict
    featIndex = featLables.index(firstStr[:firstStr.index('=')])  # 获取决策树第一层在featLables中的位置
    for key in secondDict.keys():
        if testVec[featIndex] > float(firstStr[firstStr.index('=') + 1:]):
            if type(secondDict['大于']).__name__ == 'dict':
                classLabel = classify(secondDict['大于'], featLables, testVec)
            else:
                classLabel = secondDict['大于']
            return classLabel
        else:
            if type(secondDict['小于']).__name__ == 'dict':
                classLabel = classify(secondDict['小于'], featLables, testVec)
            else:
                classLabel = secondDict['小于']
            return classLabel
```

```
# 保存决策树
def storeTree(inputTree, filename):
    import pickle
    fw = open(filename, 'wb')
    pickle.dump(inputTree, fw)
    fw.close()


# 读取决策树
def grabTree(filename):
    import pickle
    fr = open(filename, 'rb')
    return pickle.load(fr)
```

对模型进行评估代码：

```
def calConfuMatrix(myTree, label, test_dataSet):

    matrix = {1.0: {1.0: 0 , 2.0: 0 , 3.0: 0},
              2.0: {1.0: 0 , 2.0: 0 , 3.0: 0},
              3.0: {1.0: 0 , 2.0: 0 , 3.0: 0} }

    for data in test_dataSet:
        predict = classify(myTree, label, data)  # 对测试数据进行测试，predict为预测的分类
        actual = data[-1]  # actual 为实际的分类
        matrix[actual][predict] += 1  # 填充confusion matrix
    return matrix
# 查准率
def precision(classes, matrix):
    precisionDict = {'C=1': 0, 'C=2': 0, 'C=3': 0}
    for classItem in classes:
        true_predict_num = matrix[classItem][classItem]  # 准确预测数量
        all_predict_num = 0
        for temp_class_item in classes:
            all_predict_num += matrix[temp_class_item][classItem]
        precisionDict[classItem] = round(true_predict_num / all_predict_num, 2)
    return precisionDict


# 查全率
def recall(classes, matrix):
    recallDict = {'C=1': 0, 'C=2': 0, 'C=3': 0}
    for classItem in classes:
        true_predict_num = matrix[classItem][classItem]  # 准确预测数量
        all_predict_num = 0
        for temp_class_item in classes:
            all_predict_num += matrix[classItem][temp_class_item]
        recallDict[classItem] = round(true_predict_num / all_predict_num, 2)
    return recallDict
```

```
# 展示结果
def showResult(precisionValue, recallValue, classes):
    print('\t\t', '查准率', '\t', '查全率', '\t', "F1")
    for classItem in classes:
        print(classItem, '\t', precisionValue[classItem], '\t', recallValue[classItem], '\t',
              2 / (1 / precisionValue[classItem] + 1 / recallValue[classItem]))
```

【CART 算法实现-采取 sklearn 掉包方法】：

安装相关模块，导入所需库，（安装 graphviz 需要）设置好相关环境变量，如图 4 所示：

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
import xlrd
from sklearn.model_selection import train_test_split
from six import StringIO
from sklearn.tree import export_graphviz
import pydotplus
import os
os.environ['PATH'] = os.environ['PATH'] + (';D:\\科目学习类\\pythonhomeworkCODE\\exercise3\\Graphviz\\bin\\')
```

图 4

进行 cart 决策树模型搭建，并利用 graphviz 使决策树可视化，使用决定系数 R^2 是来衡量回归的好坏

在统计学中，**决定系数**反映了因变量 y 的波动，有多少百分比能被自变量 x （用机器学习的术语来说， x 就是特征）的波动所描述。简单来说，该参数可以用来判断统计模型对数据的拟合能力（或说服力）。

如图 5 所示：

```
df = pd.DataFrame(feature_train, columns=labels)

# CART分类树，基尼系数特征选择
cart_tree = DecisionTreeClassifier(criterion='gini').fit(df, target_train)

dot_data = StringIO()
export_graphviz(cart_tree, out_file=dot_data, feature_names=labels,filled=True, rounded=True,
                special_characters=True, precision=2)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('cart.png')

#线性回归的score函数返回的是：对预测结果计算出的决定系数R^2
scores_test = cart_tree.score(feature_test, target_test)
print("决定系数R^2=", scores_test)
```

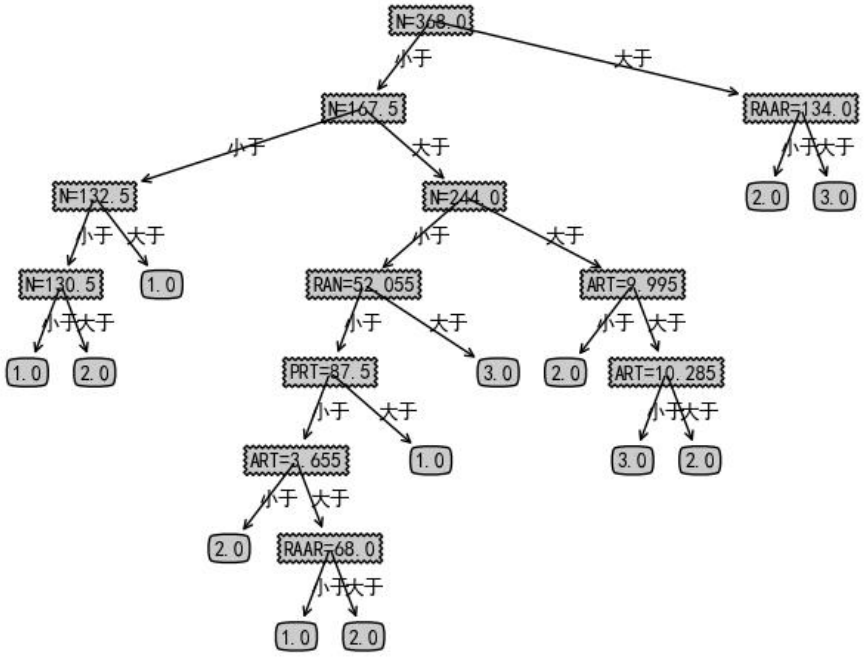
图 5

【两种算法效果对比与优缺点】：

效果对比：

C4.5:

决策树模型：



评估参数结果：

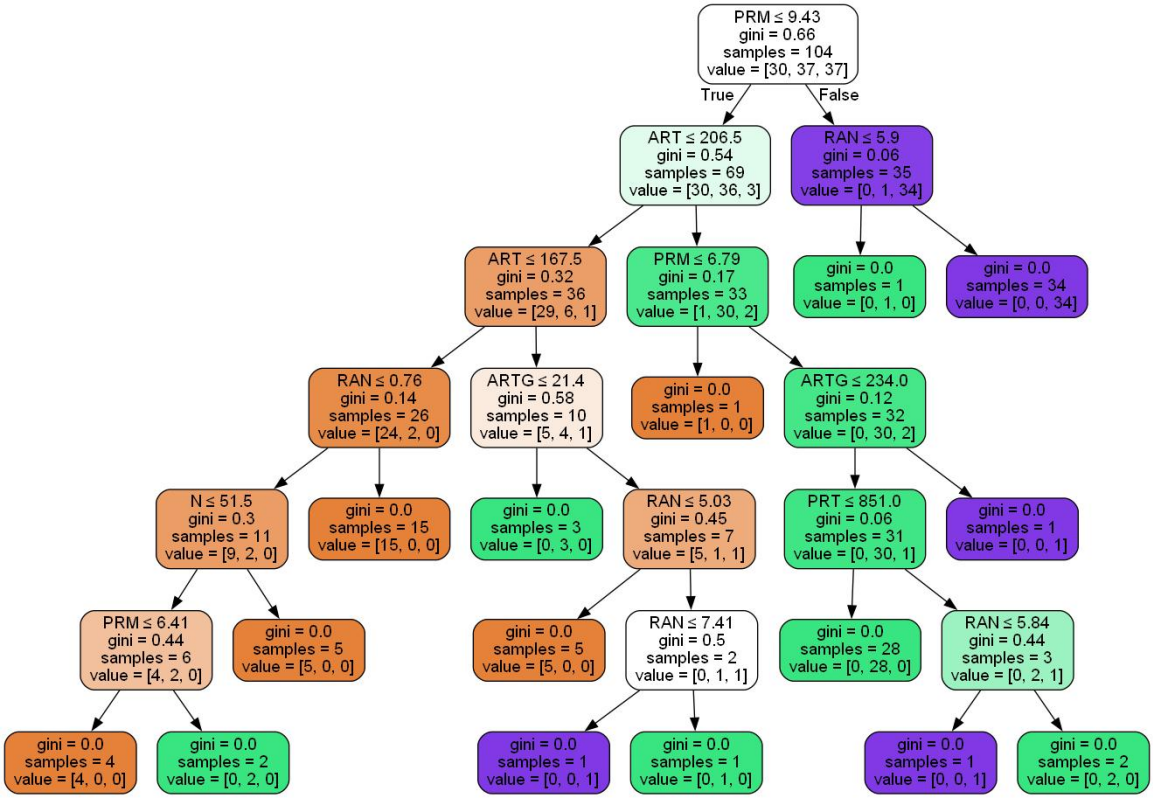
```
03C4.5
D:\developlAPP\python3.10.4\python.exe D:\科目学习类\pythonhomeworkCODE\exercise3\03C4.5.py
混淆矩阵：  {1.0: {1.0: 15, 2.0: 2, 3.0: 0}, 2.0: {1.0: 3, 2.0: 7, 3.0: 1}, 3.0: {1.0: 0, 2.0: 1, 3.0: 16}}
          查准率    查全率    F1
1.0       0.83     0.88     0.8542690058479532
2.0       0.7      0.64     0.6686567164179104
3.0       0.94     0.94     0.9400000000000001
```

为了方便理解，将混淆矩阵用 excel 表格形式呈现，如下表：

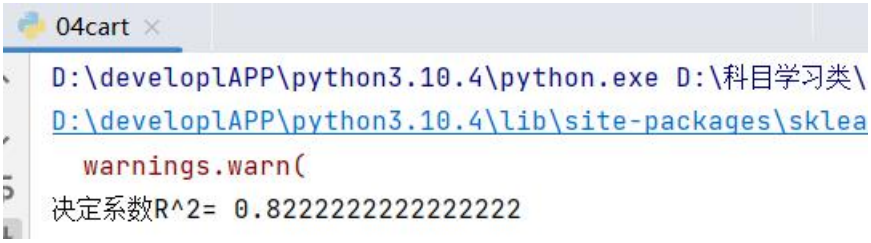
混淆矩阵		预测值		
		1	2	3
真实值	1	15	2	0
	2	3	7	1
	3	0	1	16

Cart:

决策树模型：



评估参数结果：



优缺点（参考-《机器学习》周志华）：

①从样本量考虑的话，小样本建议 C4.5、大样本建议 CART。C4.5 处理过程中需对数据集进行多次扫描排序，处理成本耗时较高，而 CART 本身是一种大样本的统计方法，小样本处理下泛化误差较大。

②C4.5 使用信息增益率克服信息增益(ID3)的缺点，偏向于特征值小的特征，CART 使用基尼指数克服 C4.5 需要求 log 的巨大计算量，偏向于特征值较多的特征。

③C4.5 剪枝策略可以再优化；C4.5 用的是多叉树，用二叉树效率更高；C4.5 只能用于分类；C4.5 使用的熵模型拥有大量耗时的对数运算，连续值还有排序运算；C4.5 在构造树的过程中，对数值属性值需要按照其大小进行排序，从中选择一个分割点，所以只适合于能够驻留于内存的数据集，当训练集大得无法在内存容纳时，程序无法运行。

④CART 为二叉树，运算速度快；CART 既可以分类也可以回归；CART 使用 Gini 系数作为变量的不纯度量，减少了大量的对数运算；CART 采用代理测试来估计缺失值，而 C4.5 以不同概率划分到不同节点中；CART 采用“基于代价复杂度剪枝”方法进行剪枝，而 C4.5 采用悲观剪枝方法。

三、总结（心得体会）

①本次实验主要对机器学习经典算法决策树进行回顾，与对实验数据进行相应分析评估，并对于不同决策树算法进行比较。

②整体实验过程中，**对于使用 python 对初始的数据处理掌握不到位，导致初期频频报错（处理不同格式文件或数据）**，后期**绘制决策树也学习到有多种方法**，如：导入自定义包（如：**treePlotter**），自定义函数，直接导入相关模块等。

③最终分析比较结果，认为 cart 效果优于 c4.5，但由于输入数据格式转换出现问题（list 与数组之间格式转换频频出错），因此进行两种算法效果比较时，采取了不同的比较参数进行对比，相对实验而言缺失严谨性。