**智能信息系统综合实践**

**实验报告**

|  |  |
| --- | --- |
| **题 目：** | **第三次作业\_决策树** |
| **年 级：** | **2021** |
| **专 业：** | **软件工程** |
| **学 号：** | **2021117405** |
| **姓 名：** | **孙潇桐** |

目录

[一、 题目 3](#_Toc160606789)

[二、 解题步骤 3](#_Toc160606790)

[1. 思考&代码 3](#_Toc160606791)

[数据的处理和划分： 3](#_Toc160606792)

[树的储存和搜索： 6](#_Toc160606793)

[数据集的加载和划分： 6](#_Toc160606794)

[决策树的测试和结果展示： 8](#_Toc160606795)

[使用sklearn包得到CART决策树： 9](#_Toc160606796)

[2. 结果 9](#_Toc160606797)

[3. 完整代码 10](#_Toc160606798)

[C4.5.py 10](#_Toc160606799)

[cart.py 13](#_Toc160606800)

[utils.py 13](#_Toc160606801)

[三、 总结 16](#_Toc160606802)

# 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。

# 解题步骤

## 思考&代码

**这部分将思路和代码结合，后面有完整的代码。**

### 数据的处理和划分：

C4.5的原理就是先计算用不同属性划分的数据集对比原来属性的信息增益率，在这些信息增益率中选择信息增益率最高的属性来对数据集进行划分。其中最重要的信息增益率的计算公式为：

但是这次的数据只有连续值，所以为了实现方便，我将上面的公式做了如下的简化。因为连续值属性使用某个中值来划分数据集，所以每次只会将数据集划分为两个，于是我们可以得知需要构建的这个决策树是一个二叉树。所以计算公式的实现可以特化为：其中表示小于选择的中值的数据集，反之。

根据上面的公式可以实现出以下的代码：

# 计算信息熵

def entropy(y):

    # 数出每种标签的个数

    \_, counts = np.unique(y, return\_counts=True)

    # 计算占比

    prob = counts / len(y)

    # 得到信息熵

    return -np.sum(prob \* np.log2(prob))

# 计算信息增益率

def information\_gain\_ratio(x, y, feature, threshold, pre\_entropy):

    # 根据当前选择的中值划分数据集

    left\_indices = x[:, feature] < threshold

    right\_indices = ~left\_indices

    # 计算比中值小部分的熵和占比

    left\_entropy = entropy(y[left\_indices])

    left\_weight = np.sum(left\_indices) / len(y)

    # 计算比中值大部分的熵和占比

    right\_entropy = entropy(y[right\_indices])

    right\_weight = np.sum(right\_indices) / len(y)

    # 计算信息增益

    information\_gain = pre\_entropy - (

        left\_weight \* left\_entropy + right\_weight \* right\_entropy

    )

    # 如果有一边的比例是1（属性值均相等）需要特判以防止计算 log(0)

    left = left\_weight if left\_weight > 0 else 1

    right = right\_weight if right\_weight > 0 else 1

    # 计算并返回信息增益率

    iv = -np.sum(left \* np.log2(left) + right \* np.log2(right))

    if iv == 0:

        return -1

    return information\_gain / iv

对于这次的软木塞数据集，经过我的观察均为**连续数据**。于是就需要实现C4.5对连续数据的处理逻辑：将当前的值排序，然后计算中值，再根据中值计算信息增益率。下面是这部分的实现代码：

# 对每个属性尝试划分

    for feature\_idx in range(len(col\_name)):

        # 取出当前计算的属性的列，保存为新数组

        cur\_attr\_list = x[:, feature\_idx].copy()

        cur\_attr\_list.sort()

        thresholds = []

        # 在对数组排序之后存下两两之间的均值

        for pre, now in zip(cur\_attr\_list, cur\_attr\_list[1:]):

            thresholds.append((now + pre) / 2)

        thresholds = np.array(thresholds)

        # 对均值进行去重，简化后面的计算

        thresholds = np.unique(thresholds)

        for threshold in thresholds:

            # 对每个中值计算信息增益率

            gain\_ratio = information\_gain\_ratio(

                x, y, feature\_idx, threshold, pre\_entropy

            )

            # 保存最好的信息增益率

            if gain\_ratio >= best\_gain\_ratio:

                best\_gain\_ratio = gain\_ratio

                best\_feature\_idx = feature\_idx

                best\_threshold = threshold

在求出最好的信息增益率之后就可以对数据集进行划分，然后根据当前求出的信息增益率的阈值，将数据集分为左右子树：

def build(x, y):

    # 如果当前数据集只有一个标签值，选择当前节点为叶子节点

    if len(np.unique(y)) == 1:

        return Node(type=y[0])

    best\_gain\_ratio = 0

    best\_feature\_idx = None

    best\_threshold = None

    # 计算划分前的信息熵

    pre\_entropy = entropy(y)

    # 对每个属性尝试划分

    for feature\_idx in range(len(col\_name)):

        # 这段代码在上面贴过，这里不再重复……

    # 根据阈值筛选出左右子树的数据集

    left\_indices = x[:, best\_feature\_idx] < best\_threshold

    right\_indices = ~left\_indices

    # 如果阈值左边有数据的话递归建立左子树

    if np.sum(left\_indices) > 0:

        left\_subtree = build(x[left\_indices], y[left\_indices])

    else:

        left\_subtree = None

    # 如果阈值右边有数据的话递归建立右子树

    if np.sum(right\_indices) > 0:

        right\_subtree = build(x[right\_indices], y[right\_indices])

    else:

        right\_subtree = None

    # 左右子树建立完成之后将引用保存在当前节点

    return Node(

        feature=best\_feature\_idx,

        threshold=best\_threshold,

        left=left\_subtree,

        right=right\_subtree,

    )

### 树的储存和搜索：

树是由节点组成的，所以将数据储存在结点类中就行。因为这个数据集中所有的参数都是连续值，根据C4.5的算法，将数据用某个中值划分成两半，所以只会有左右子树。同时查找也只需要使用类似二叉搜索树的方法，根据节点中储存的阈值将样本导向左子树或者右子树就行，下面是我对节点的实现：

class Node:

    """

    决策树的节点

    属性:

        feature (int): 该节点所选择的特征的索引

        threshold (float):  特征划分的阈值

        left (Node): 左子树引用

        right (Node): 右子树引用

        type (int or None): 如果是叶子节点就是最终的类别, 不是则是None

    """

    def \_\_init\_\_(self, feature=None, threshold=None, left=None, right=None, type=None):

        self.feature = feature

        self.threshold = threshold

        self.left = left

        self.right = right

        self.type = type

    def predict(self, attrs):

        # 属性值可能是一个列表

        if len(np.shape(attrs)) > 1:

            attrs = attrs[0]

        # 到了叶子节点，返回叶子节点的值

        if self.type is not None:

            return self.type

        # 根据节点中储存的阈值决定遍历左或右子树

        if attrs[self.feature] < self.threshold:

            return self.left.predict(attrs)

        else:

            return self.right.predict(attrs)

### 数据集的加载和划分：

因为这些数据是要用两次：先训练我的C4.5决策树再训练CART决策树，所以我将数据的加载和划分分到了另一个文件，只要在需要使用函数的文件头部import就能方便的使用，避免了冗余的代码，下面是我的实现：

# 用于从文件导入数据

def load\_data(file\_loc):

    data = pd.read\_excel(file\_loc, sheet\_name="Data", header=0)

    col\_name = data.columns.to\_list()[2:]

    col\_name[1] = "N"

    # print(col\_name)

    data\_set = []

    # 按行读取文件

    for \_, row in data.iterrows():

        data\_set.append(row.to\_numpy()[1:])

    # print(data\_set)

    data\_set = np.array(data\_set)

    X = data\_set[:, 1:]  # 特征值

    Y = data\_set[:, 0]  # 标签

    tags = np.unique(Y)

    # 返回值是 特征值列表，标签列表，属性列名，标签种类列表

    return X, Y, col\_name, tags

# 用于将数据集分为训练集和测试集

def split(X, Y, test\_size, seed):

    # 设定给定的种子，保证每次随机的结果是相同的

    np.random.seed(seed)

    # 随机生成索引

    indices = np.random.permutation(len(X))

    test\_set\_size = int(len(X) \* test\_size)

    test\_indices = indices[:test\_set\_size]

    train\_indices = indices[test\_set\_size:]

    x\_train = np.array([X[i] for i in train\_indices])

    x\_test = np.array([X[i] for i in test\_indices])

    y\_train = np.array([Y[i] for i in train\_indices])

    y\_test = np.array([Y[i] for i in test\_indices])

    # 返回 训练集的特征值列表和标签，测试集的特征值列表和标签

    return x\_train, y\_train, x\_test, y\_test

### 决策树的测试和结果展示：

因为需要对比两种决策树的结果，所以我写了一个可以兼容我自己训练的决策树和使用sklearn包得到决策树的测试函数，测试函数可以输出混淆矩阵，查准率，查全率和F1\_Score。然后我又写了一个展示函数将这些结果格式化之后展示出来，下面是这两个函数的实现：

# 测试函数

def test(tree, x, y, tags):

    tag\_num = len(tags)

    tag\_map = dict()

    for idx, tag in enumerate(tags):

        tag\_map[tag] = idx

    # 多分类混淆矩阵，行为预测值，列为真实值

    confusion\_matrix = np.zeros((tag\_num, tag\_num), dtype=int)

    for now, tag in zip(x, y):

        predict\_tag = tree.predict([now])

        if len(np.shape(predict\_tag)) > 0:

            predict\_tag = predict\_tag[0]

        confusion\_matrix[tag\_map[predict\_tag], tag\_map[tag]] += 1

    # 查准率

    precision = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=1)

    # 查全率

    recall = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=0)

    # F1

    f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

    return confusion\_matrix, precision, recall, f1

# 展示结果（混淆矩阵，查准率，查全率和F1\_Score)

def display(cm, pc, rc, f1, tags):

    print("\t\t混淆矩阵")

    print("\t\t\t真实值")

    str = "  \t"

    for x in tags:

        str += f"\t{x}"

    print(str)

    s\_idx = 0

    str = "预测值"

    for idx, tag in enumerate(tags):

        now = str[s\_idx] if idx < len(str) else "  "

        s\_idx += 1

        now += f"\t{tag}"

        for i in cm[idx]:

            now += f"\t{i}"

        print(now)

    print()

    print("类型\t查准率\t查全率\tF1\_Score")

    for idx, tag in enumerate(tags):

        print(f"{tag}\t{pc[idx]:.4}\t{rc[idx]:.4}\t{f1[idx]:.4}")

### 使用sklearn包得到CART决策树：

因为需要对比两种决策树的结果，所以我使用sklearn包训练了一个CART决策树，然后使用我的测试函数进行测试，我会在后面展示两个模型的测试结果，下面是我使用这个包的过程：

from sklearn.tree import DecisionTreeClassifier

# 我写的工具类

from utils import load\_data, split, test, display

X, Y, col\_name, tags = load\_data("data/cork\_stoppers.xlsx")

x\_train, y\_train, x\_test, y\_test = split(X, Y, 0.3, 42)

# 创建CART模型，使用基尼系数

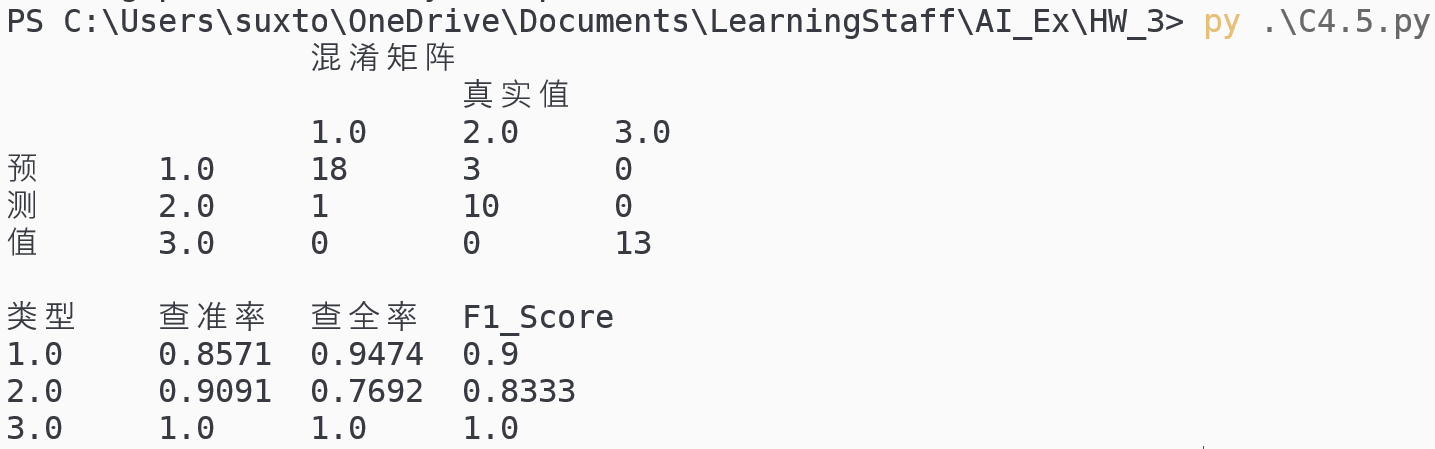
tree = DecisionTreeClassifier(criterion="gini")

# 训练模型

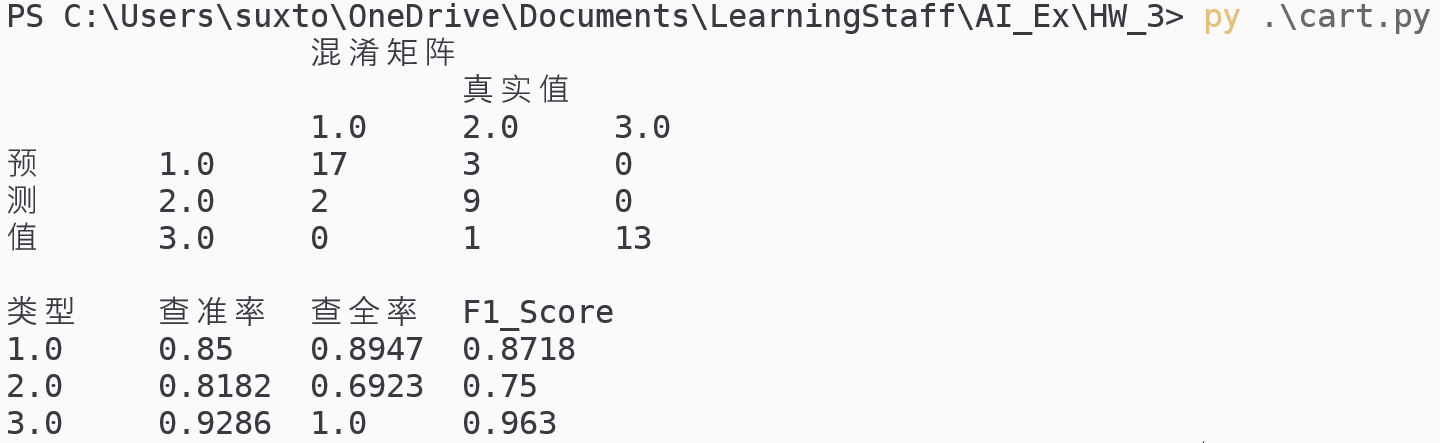
tree.fit(x\_train, y\_train)

## 结果

使用我的C4.5决策树的测试结果：



使用我的CART决策树的测试结果：



从最常参考的F1\_Score来看似乎是C4.5更加优秀，C4.5每一个类型的F1\_Score都大于CART的结果，两个算法优缺点的总结在文档最后。

## 完整代码

### C4.5.py

import numpy as np

from utils import load\_data, split, test, display

X, Y, col\_name, tags = load\_data("data/cork\_stoppers.xlsx")

x\_train, y\_train, x\_test, y\_test = split(X, Y, 0.3, 42)

class Node:

    """

    决策树的节点

    属性:

        feature (int): 该节点所选择的特征的索引

        threshold (float):  特征划分的阈值

        left (Node): 左子树引用

        right (Node): 右子树引用

        type (int or None): 如果是叶子节点就是最终的类别, 不是则是None

    """

    def \_\_init\_\_(self, feature=None, threshold=None, left=None, right=None, type=None):

        self.feature = feature

        self.threshold = threshold

        self.left = left

        self.right = right

        self.type = type

    def predict(self, attrs):

        # 属性值可能是一个列表

        if len(np.shape(attrs)) > 1:

            attrs = attrs[0]

        # 到了叶子节点，返回叶子节点的值

        if self.type is not None:

            return self.type

        # 根据节点中储存的阈值决定遍历左或右子树

        if attrs[self.feature] < self.threshold:

            return self.left.predict(attrs)

        else:

            return self.right.predict(attrs)

# 计算信息熵

def entropy(y):

    # 数出每种标签的个数

    \_, counts = np.unique(y, return\_counts=True)

    # 计算占比

    prob = counts / len(y)

    # 得到信息熵

    return -np.sum(prob \* np.log2(prob))

# 计算信息增益率

def information\_gain\_ratio(x, y, feature, threshold, pre\_entropy):

    # 根据当前选择的中值划分数据集

    left\_indices = x[:, feature] < threshold

    right\_indices = ~left\_indices

    # 计算比中值小部分的熵和占比

    left\_entropy = entropy(y[left\_indices])

    left\_weight = np.sum(left\_indices) / len(y)

    # 计算比中值大部分的熵和占比

    right\_entropy = entropy(y[right\_indices])

    right\_weight = np.sum(right\_indices) / len(y)

    # 计算信息增益

    information\_gain = pre\_entropy - (

        left\_weight \* left\_entropy + right\_weight \* right\_entropy

    )

    # 如果有一边的比例是1（当前选择的属性值均相等）需要特判以防止计算 log(0)

    left = left\_weight if left\_weight > 0 else 1

    right = right\_weight if right\_weight > 0 else 1

    # 计算并返回信息增益率

    iv = -np.sum(left \* np.log2(left) + right \* np.log2(right))

    if iv == 0:

        return -1

    return information\_gain / iv

def build(x, y):

    # 如果当前数据集只有一个标签值，选择当前节点为叶子节点

    if len(np.unique(y)) == 1:

        return Node(type=y[0])

    best\_gain\_ratio = 0

    best\_feature\_idx = None

    best\_threshold = None

    # 计算划分前的信息熵

    pre\_entropy = entropy(y)

    # 对每个属性尝试划分

    for feature\_idx in range(len(col\_name)):

        # 取出当前计算的属性的列，保存为新数组

        cur\_attr\_list = x[:, feature\_idx].copy()

        cur\_attr\_list.sort()

        thresholds = []

        # 在对数组排序之后存下两两之间的均值

        for pre, now in zip(cur\_attr\_list, cur\_attr\_list[1:]):

            thresholds.append((now + pre) / 2)

        thresholds = np.array(thresholds)

        # 对均值进行去重，简化后面的计算

        thresholds = np.unique(thresholds)

        for threshold in thresholds:

            # 对每个中值计算信息增益率

            gain\_ratio = information\_gain\_ratio(

                x, y, feature\_idx, threshold, pre\_entropy

            )

            # 保存最好的信息增益率

            if gain\_ratio >= best\_gain\_ratio:

                best\_gain\_ratio = gain\_ratio

                best\_feature\_idx = feature\_idx

                best\_threshold = threshold

    # 根据阈值筛选出左右子树的数据集

    left\_indices = x[:, best\_feature\_idx] < best\_threshold

    right\_indices = ~left\_indices

    # 如果阈值左边有数据的话递归建立左子树

    if np.sum(left\_indices) > 0:

        left\_subtree = build(x[left\_indices], y[left\_indices])

    else:

        left\_subtree = None

    # 如果阈值右边有数据的话递归建立右子树

    if np.sum(right\_indices) > 0:

        right\_subtree = build(x[right\_indices], y[right\_indices])

    else:

        right\_subtree = None

    # 左右子树建立完成之后将引用保存在当前节点

    return Node(

        feature=best\_feature\_idx,

        threshold=best\_threshold,

        left=left\_subtree,

        right=right\_subtree,

    )

tree = build(x\_train, y\_train)

cm, pc, rc, f1 = test(tree, x\_test, y\_test, tags)

display(cm, pc, rc, f1, tags)

### cart.py

from sklearn.tree import DecisionTreeClassifier

# 我写的工具类

from utils import load\_data, split, test, display

X, Y, col\_name, tags = load\_data("data/cork\_stoppers.xlsx")

x\_train, y\_train, x\_test, y\_test = split(X, Y, 0.3, 42)

# 创建CART模型

tree = DecisionTreeClassifier(criterion="gini")

# 训练模型

tree.fit(x\_train, y\_train)

cm, pc, rc, f1 = test(tree, x\_test, y\_test, tags)

display(cm, pc, rc, f1, tags)

### utils.py

import pandas as pd

import numpy as np

# 用于从文件导入数据

def load\_data(file\_loc):

    data = pd.read\_excel(file\_loc, sheet\_name="Data", header=0)

    col\_name = data.columns.to\_list()[2:]

    col\_name[1] = "N"

    # print(col\_name)

    data\_set = []

    # 按行读取文件

    for \_, row in data.iterrows():

        data\_set.append(row.to\_numpy()[1:])

    # print(data\_set)

    data\_set = np.array(data\_set)

    X = data\_set[:, 1:]  # 特征值

    Y = data\_set[:, 0]  # 标签

    tags = np.unique(Y)

    # 返回值是 特征值列表，标签列表，属性列名，标签种类列表

    return X, Y, col\_name, tags

# 用于将数据集分为训练集和测试集

def split(X, Y, test\_size, seed):

    # 设定给定的种子，保证每次随机的结果是相同的

    np.random.seed(seed)

    # 随机生成索引

    indices = np.random.permutation(len(X))

    test\_set\_size = int(len(X) \* test\_size)

    test\_indices = indices[:test\_set\_size]

    train\_indices = indices[test\_set\_size:]

    x\_train = np.array([X[i] for i in train\_indices])

    x\_test = np.array([X[i] for i in test\_indices])

    y\_train = np.array([Y[i] for i in train\_indices])

    y\_test = np.array([Y[i] for i in test\_indices])

    # 返回 训练集的特征值列表和标签，测试集的特征值列表和标签

    return x\_train, y\_train, x\_test, y\_test

# 测试函数

def test(tree, x, y, tags):

    tag\_num = len(tags)

    tag\_map = dict()

    for idx, tag in enumerate(tags):

        tag\_map[tag] = idx

    # 多分类混淆矩阵，行为预测值，列为真实值

    confusion\_matrix = np.zeros((tag\_num, tag\_num), dtype=int)

    for now, tag in zip(x, y):

        predict\_tag = tree.predict([now])

        if len(np.shape(predict\_tag)) > 0:

            predict\_tag = predict\_tag[0]

        confusion\_matrix[tag\_map[predict\_tag], tag\_map[tag]] += 1

    # 查准率

    precision = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=1)

    # 查全率

    recall = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=0)

    # F1

    f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

    return confusion\_matrix, precision, recall, f1

# 展示结果（混淆矩阵，查准率，查全率和F1\_Score)

def display(cm, pc, rc, f1, tags):

    print("\t\t混淆矩阵")

    print("\t\t\t真实值")

    str = "  \t"

    for x in tags:

        str += f"\t{x}"

    print(str)

    s\_idx = 0

    str = "预测值"

    for idx, tag in enumerate(tags):

        now = str[s\_idx] if idx < len(str) else "  "

        s\_idx += 1

        now += f"\t{tag}"

        for i in cm[idx]:

            now += f"\t{i}"

        print(now)

    print()

    print("类型\t查准率\t查全率\tF1\_Score")

    for idx, tag in enumerate(tags):

        print(f"{tag}\t{pc[idx]:.4}\t{rc[idx]:.4}\t{f1[idx]:.4}")

# 总结

两种算法的优缺点总结：

**C4.5算法优缺点**：

(1) 产生的规则易于理解，准确率较高（在我的实验中），实现简单。

(2) 在建树的过程中需要多次顺序扫描和排序数据，训练的效率较低。

(3) 信息增益率更偏向于选择取值更少的属性。

**CART算法优缺点**：

(1) 使用一组特征值而不是一个特征值，分类更加合理。

(2) 因为基于基尼指数，所以容易被噪声干扰。

去年在学习web数据挖掘的时候我也自己实现过一个C4.5决策树，但是当时老师给的数据没有连续值，所以当时只实现了分类离散值的功能。这次为了便于实现，我实现了一个只支持连续值的决策树，等有机会我就把这两个决策树结合一下，实现出比较完整的C4.5决策树。

这次的实验因为我不熟悉NumPy，所以走了很多弯路，比如我曾经直接对数组切片排序导致数据集的混乱，但是这些问题都被我一一解决，解决这些问题的经历已经转化为我的经验，这次试验使我受益匪浅。