# 西北大学

**智能信息系统综合实践**

**实验报告**

| | |
|---|---|
| **题　目:** | **第二次作业** |
| **年　级:** | 2020 |
| **专　业:** | 软件工程 |
| **姓　名:** | 刘航 |

## 一、 题目（原题目）

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）

### 问题一:

划分训练集合和测试集合

采用分层抽样的方法，将整体数据集合的 1/3 作为测试数据集

```python
def train_and_test_divide(samples_set):
    # 采用分层抽样的方法
    # 1/3作为测试
    kinds = samples_set.loc[:, 'C']
    kinds = list(kinds)
    # print(kinds)
    # print(type(kinds))
    kinds_set = set(kinds)
    kinds_num = len(kinds_set)
    kinds_dict = {}
    for item in kinds_set:
        # print(item)
        kinds_dict.update({item: kinds.count(item)})
    test_set = pd.DataFrame(data=None, columns=samples_set.columns)
    train_set = samples_set
    pianyi = 0
    print(samples_set)
    for cla in kinds_set:
        num = int(1 / 3 * kinds_dict[cla])

        k = random.sample(range(pianyi, pianyi + kinds_dict[cla]), num)

        for i in k:
            train_set = train_set.drop(index=i)
            test_set.loc[len(test_set), :] = samples_set.loc[i, :]
        pianyi += kinds_dict[cla]
    train_set = train_set.reset_index(drop=True)
    test_set = test_set.reset_index(drop=True)
    return train_set, test_set
```

结果：训练集和测试集合数据。

训练集合

| | C | ART | N | PRT | ARM | PRM | ARTG | NG | PRTG | RAAR | RAN |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 125 | 63 | 368 | 1.98 | 5.84 | 20.0 | 1.0 | 18 | 16.00 | 1.59 |
| 1 | 1 | 146 | 45 | 350 | 3.24 | 7.78 | 42.8 | 2.8 | 43 | 29.28 | 6.11 |
| 2 | 1 | 109 | 55 | 316 | 1.98 | 5.75 | 0.8 | 0.8 | 3 | 0.69 | 1.36 |
| 3 | 1 | 161 | 57 | 400 | 2.83 | 7.02 | 19.0 | 1.0 | 18 | 11.80 | 1.75 |
| 4 | 1 | 156 | 64 | 424 | 2.44 | 6.63 | 18.0 | 2.0 | 24 | 11.54 | 3.13 |
| .. | .. | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 97 | 3 | 556 | 134 | 1288 | 4.15 | 9.61 | 336.0 | 12.0 | 262 | 60.43 | 8.96 |
| 98 | 3 | 660 | 113 | 1320 | 5.84 | 11.68 | 410.0 | 14.0 | 338 | 62.12 | 12.39 |
| 99 | 3 | 720 | 99 | 1274 | 7.27 | 12.87 | 562.0 | 13.0 | 406 | 78.06 | 13.13 |
| 100 | 3 | 566 | 83 | 1024 | 6.82 | 12.34 | 342.0 | 13.0 | 278 | 60.42 | 15.66 |
| 101 | 3 | 474 | 93 | 1014 | 5.10 | 10.90 | 204.0 | 7.0 | 158 | 43.04 | 7.53 |

[102 rows x 11 columns]

图（1）：训练集合部分数据

测试集合

| | C | ART | N | PRT | ARM | PRM | ARTG | NG | PRTG | RAAR | RAN |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.0 | 76.0 | 40.0 | 228.0 | 1.9 | 5.7 | 0.8 | 0.8 | 3.0 | 0.99 | 1.88 |
| 1 | 1.0 | 80.0 | 42.0 | 238.0 | 1.91 | 5.67 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2 | 1.0 | 35.0 | 18.0 | 104.0 | 1.94 | 5.78 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3 | 1.0 | 136.0 | 68.0 | 392.0 | 2.0 | 5.77 | 12.0 | 1.0 | 14.0 | 8.82 | 1.47 |
| 4 | 1.0 | 81.0 | 41.0 | 250.0 | 1.98 | 6.1 | 9.0 | 1.0 | 12.0 | 11.11 | 2.44 |
| 5 | 1.0 | 81.0 | 26.0 | 196.0 | 3.12 | 7.54 | 9.8 | 1.8 | 15.0 | 12.04 | 6.73 |
| 6 | 1.0 | 88.0 | 40.0 | 246.0 | 2.2 | 6.15 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7 | 1.0 | 155.0 | 61.0 | 418.0 | 2.54 | 6.85 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8 | 1.0 | 112.0 | 54.0 | 318.0 | 2.07 | 5.89 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 9 | 1.0 | 119.0 | 53.0 | 342.0 | 2.25 | 6.45 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 10 | 1.0 | 92.0 | 50.0 | 292.0 | 1.84 | 5.84 | 12.0 | 1.0 | 14.0 | 13.04 | 2.0 |
| 11 | 1.0 | 179.0 | 89.0 | 520.0 | 2.01 | 5.84 | 21.8 | 2.8 | 29.0 | 12.15 | 3.09 |
| 12 | 1.0 | 136.0 | 66.0 | 402.0 | 2.06 | 6.09 | 9.0 | 1.0 | 12.0 | 6.62 | 1.52 |
| 13 | 1.0 | 130.0 | 55.0 | 348.0 | 2.36 | 6.33 | 24.0 | 2.0 | 28.0 | 18.46 | 3.64 |
| 14 | 1.0 | 165.0 | 63.0 | 430.0 | 2.62 | 6.83 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 15 | 1.0 | 143.0 | 44.0 | 332.0 | 3.25 | 7.55 | 30.0 | 2.0 | 32.0 | 20.98 | 4.55 |
| 16 | 2.0 | 302.0 | 88.0 | 694.0 | 3.43 | 7.89 | 88.0 | 6.0 | 94.0 | 29.14 | 6.82 |
| 17 | 2.0 | 258.0 | 61.0 | 538.0 | 4.23 | 8.82 | 97.0 | 6.0 | 102.0 | 37.6 | 9.84 |
| 18 | 2.0 | 223.0 | 64.0 | 526.0 | 3.48 | 8.22 | 27.0 | 2.0 | 30.0 | 12.11 | 3.13 |

图（2)测试集合部分数据

## 问题二：

进行 C4.5 决策树的创建，并且进行评估

**step1**:属性类和树节点类的定义

属性类说明：包含属性名字和属性类型。

name：表示的是这个属性的名词，用来区分不同属性。

type=0，表示的是离散属性，type=1 表示连续属性。

values：如果属性是一个离散属性的话，values 将保存这个属性所有可能的取值。

节点类说明：节点类是构成决策树的基本节点。

feature：每个节点都有一个特征，这个特征是进行预测时的预测样本走到这个节点的，需要选择去判断的属性。

如果这个属性为 None 则是一个叶子节点。

kinds：表示的是这个节点的类别，如果一个样本走到这里不能继续前进的话，就将 kinds 的类别作为样本

类型。

divide_value：当这个节点选择的属性是一个连续属性的时候这个值有意义。

如果这个节点的属性是连续属性，则它的值就是样本进入到这个节点是，选择

这个连续属性，并且判断应该进入那个分支的标准。

child_node:这个存储的是子树节点，如果 child_node 为空的话，则说明这是一个叶子节点。

函数 find_path(self, sample)：这是建立树后，进行树的遍历，样本路径选择的函数，通过返回 child_node 列表的

下标，表示下一次应该走那个节点。

```
class Attribute:
    name = ''
    type = 0
    values = []

class Node:
    feature = Attribute()
    kinds = 0
    divide_value = 0
    child_node = []

    def find_path(self, sample):

        if len(self.child_node) == 0:
            return -1

        # 如果这个节点使用的是离散属性进行划分的话，执行下面
        if self.feature.type == 0:
            name = self.feature.name
            value = sample[name]
            for i in range(len(self.feature.values)):
                if self.feature.values[i] == value:
                    return i
            return -1
        # 节点划分使用的是连续属性
        else:
            name = self.feature.name
            value = sample[name]
            if value <= self.divide_value:
                return 0
            else:
                return 1

```

step2：建立一个决策树。

建立树的思想：

使用递归思想建立一个决策树。

（1）函数输入参数：传入函数一个样本集合和一个属性集合。

样本集合：所有训练样本构成的集合。

属性集合：这个样本集合中可以选择作为子集划分的属性集合。

如果是一个离散属性，则样本划分时只能选择一个，在递归调用时，这个属性就不能作为划分属性了。

需要从属性集合中删除。

如果是一个连续属性，选择划分后，子集中依然可以选择这个属性作为划分属性，不能从属性集合中删除。

（2）选择那个属性作为划分属性的判断

1. 计算原始样本集合的信息熵。

   这个样本集合中有几类样本，以及每类样本的数量。

   然后使用信息熵公式计算。

```
def create_decision_tree(samples_set, attribute_set=[]):
    node = Node()
    node.child_node = []
    kinds = samples_set.loc[:, 'C']
    kinds = list(kinds)
    # print(kinds)
    # print(type(kinds))
    kinds_set = set(kinds)
    kinds_num = len(kinds_set)
    kinds_dict = {}
    for item in kinds_set:
        # print(item)
        kinds_dict.update({item: kinds.count(item)})
    max_key = max(kinds_dict, key=kinds_dict.get)
    node.feature = None
    node.kinds = max_key

# 初始集合的信息熵
    ent = 0
    for key, value in kinds_dict.items():
        pk = value / len(samples_set)
        ent -= pk * np.log2(pk)
```

2. 判读是否不能在继续进行划分了。

   如果样本集合全部属于同一类，或者属性集合为空，或者样本集合为空，都会

   导致无法继续进行划分，这里就是递归截至的条件。

```
# 属性集合为空不可以划分了
    if len(attribute_set) == 0:
        print('调用截至属性集合为空')
        return node

    # 属于同一类，没必要划分
    if kinds_num == 1:
        print('调用截至同一类')
        print(samples_set)
        print('type', node.kinds)
        return node
```

3. 计算每个属性的信息增益和信息增益率

   离散属性计算：

   如果是离散属性直接计算按照这个属性进行划分后得到的信息增益和信息增益率

   然后将它存起来，在最后作为选择标准。

```
for i in range(len(attribute_set)):

        iv = 0
        gain = ent
```

```
        # 属性是离散属性
        if attribute_set[i].type == 0:
            # 对这个属性的每一个属性值进行操作
            for j in range(len(attribute_set[i].values)):

                # 每一个属性值生成一个子集
                sub_sa = []
                sub_kind = []

                for k in range(len(samples_set)):
                    if samples_set[attribute_set[i].name] == attribute_set[i].values[j]:
                        sub_sa.append(samples_set[k])
                        sub_kind.append(kinds[k])
                # 计算每个子集的信息熵
                sub_kind_set = set(sub_kind)
                # kinds_num = len(kinds_set)
                sub_kinds_dict = {}
                for item in sub_kind_set:
                    sub_kinds_dict.update({item: list.count(item)})

                sub_ent = 0
                for key, value in sub_kinds_dict:
                    pk = value / len(sub_sa)
                    sub_ent -= pk * np.log2(pk)
                gain -= len(sub_sa) / len(samples_set) * sub_ent
                pk = len(sub_sa) / len(samples_set)
                iv -= pk * np.log2(pk)
            gain_ratio = gain / iv
            gain_ratio_sset[i] = gain_ratio

        # 连续属性
```

连续属性：

如果是连续属性，则需要在这个连续属性中选择一个值，将这个连续属性分为两部分。

将它转换成和只有两个值的离散属性类似的。

这需要将样本集中这个连续属性的取值进行排序，从小到大排序，并且需要去除重复的值。

通过这个已经排好序的序列，如果有 n 个值，那么需要选取 n-1 个分界线，每个分界线是连续

两个值的中位数。

计算每个分界线将样本集合划分成两部分的信息增益。选取可以使得信息增益最大的分界线。

**注意：这里不能选择使信息增益率最大的分界线，只能选择信息增益最大的**

在找到最好的分界线后，计算出以这个分解线划分后的信息增益和信息增益率。

```

        # 连续属性
        else:
            #对划分属性值进行排序
            samples_set_copy = samples_set.sort_values(axis=0, by=attribute_set[i].name)
            samples_set_copy = samples_set_copy.reset_index(drop=True)
            huafen_point = set(samples_set_copy.loc[:, attribute_set[i].name].tolist())
            huafen_point = list(huafen_point)
            huafen_point.sort()
            #存放选择最优划分边界的地方
            gain_set = np.zeros(len(huafen_point) - 1)
            gain_ratio_set = np.zeros(len(huafen_point) - 1)
            divede_value_set = np.zeros(len(huafen_point) - 1)

            for j in range(len(huafen_point) - 1):
                # print('v2=',samples_set_copy.loc[j + 1,attribute_set[i].name])
                divede_value = (huafen_point[j] + huafen_point[j + 1]) / 2
                # print(j,divede_value)
```

```
                divede_value_set[j] = divede_value
                for k in range(len(samples_set_copy)):
                    if samples_set_copy.loc[k, attribute_set[i].name] > divede_value:
                        break
                #小于划分值的子集
                sub_sa_left = samples_set_copy[:k]
                # sub_kind_left = kinds[:j+1]
                #计算子集中各个类别的个数
                kinds = sub_sa_left.loc[:, 'C']
                kinds = list(kinds)
                kinds_set = set(kinds)

                # print('jaijsfaf',kinds_set)
                kinds_dict = {}
                for item in kinds_set:
                    kinds_dict.update({item: kinds.count(item)})
                sub_ent_left = 0
                #计算子集的信息熵
                for key, value in kinds_dict.items():
                    pk = value / len(sub_sa_left)
                    sub_ent_left -= pk * np.log2(pk)
                # print(sub_ent_left)
                #大于划分值的子集
                sub_sa_right = samples_set_copy[k:]

                # print('left')
                # print(sub)
                kinds = sub_sa_right.loc[:, 'C']
                kinds = list(kinds)
                kinds_set = set(kinds)
                kinds_dict = {}
                for item in kinds_set:
                    kinds_dict.update({item: kinds.count(item)})
                #计算子集的信息熵
                sub_ent_right = 0
                for key, value in kinds_dict.items():
                    pk = value / len(sub_sa_right)
                    sub_ent_right -= pk * np.log2(pk)
                # print(sub_ent_right)
                pk_left = len(sub_sa_left) / len(samples_set_copy)
                # print('pk_l',pk_left)

                pk_right = len(sub_sa_right) / len(samples_set_copy)
                # print('pk_r', pk_right)
                # print('ent=',ent)
                #计算选取这个值作为划分边界的信息增益和信息增益率
                gain = ent - pk_left * sub_ent_left - pk_right * sub_ent_right
                if gain < 0.01:
                    gain = 0

                iv = -pk_left * np.log2(pk_left) - pk_right * np.log2(pk_right)

                gain_ratio = gain / iv
                gain_ratio_set[j] = gain_ratio
                gain_set[j] = gain

            # 找划分边界的时候使用信息增益，选择属性的时候使用信息增益率
            # print('***********')
            # print(gain_set)
            gain_set = gain_set.tolist()
            # print(gain_set)
            # print('***********')

            #这是连续属性取值全部相等的情况，此时找不到划分边界
            if len(gain_set) == 0:
                max_gain = 0
                max_gain_ratio = 0
                divide[i] = samples_set.loc[0, attribute_set[i].name]
```

```
            #可以找到划分边界的情况，找出使得这个属性的信息增益最大的分界线
            else:
                max_gain = np.max(gain_set)
                value_index = gain_set.index(max_gain)
                max_gain_ratio = gain_ratio_set[value_index]
                divide[i] = divede_value_set[value_index]

            gain_ratio_sset[i] = max_gain_ratio
            gain_sset[i] = max_gain
            # 增益率最大时的分界线
```

4. 找到最优划分属性。

   C4.5 找最优属性的思想是，先找出信息增益高于平均信息增益的属性。

   然后从这些属性中找到使得信息增益率最大的属性。它作为最优划分属性。

   信息增益会对取值数较多的属性有偏好。

   信息增益率对取值较少的属性有偏好。

   结合二者会更优。

```
# 找出所有属性中可以得到最大增益率的属性
# 找出所有属性中可以得到最大增益率的属性
    #找到超过半数的信息增益的属性
    more_half_gain = []
    more_half_gain_ratio = []
    mean_gain = np.mean(gain_sset)
    for i in range(len(gain_sset)):
        if gain_sset[i] > mean_gain:
            more_half_gain.append(gain_sset[i])
            more_half_gain_ratio.append(gain_ratio_sset[i])

    # 超过半数的信息增益的属性为空，不能划分，说明任意属性划分都不能提示信息增益，所以不用划分
    if len(more_half_gain) == 0:
        return node

    #找出最好的划分属性
    best_gain_ratio = np.max(more_half_gain_ratio)
    best_a_index = gain_ratio_sset.index(best_gain_ratio)
    best_attribute = attribute_set[best_a_index]
    print('划分选择', divide)
    print('选择的划分属性为:', best_attribute.name)
    print(gain_ratio_sset)
    print('增益率为:', best_gain_ratio)
    print('连续属性的划分边界为：', divide[best_a_index])
    print('这个节点的类别是：', node.kinds)
    node.feature = best_attribute
    # 选择其中一个属性后，进行子集划分和子树的建立
    # 最终选择了离散属性
```

5. 构建子树。

   **离散属性**：如果最优划分属性是一个离散属性。

   那么将样本集合按照属性可取的值划分相应的子集。

   然后将这个离散属性从属性集合中删除。它不能作为后续节点的划分属性了。

   然后递归调用创建决策树的函数。

   可以生成和离散属性取值数量相等的子树。

   将这些子树加入到这个节点的子节点中

   **连续属性**：如果是连续属性。

   则按照连续属性的分界线将样本集合划分为两个子集。

如果子集不为空，递归调用创建决策树函数，以子集为样本构建子树。

然后将子树根节点加入到这个节点的子节点中。

如果为空，创建一个叶节点，直接加入到子节点集合中。

```
    # 选择其中一个属性后，进行子集划分和子树的建立
    # 最终选择了离散属性
    if best_attribute.type == 0:

        attribute_set_copy = attribute_set
        attribute_set_copy.remove(best_attribute)
        for a_value in best_attribute.values:

            sub_sa = []
            sub_kind = []
            for i in range(len(samples_set)):
                if samples_set[i].get(best_attribute) == a_value:
                    sub_sa.append(samples_set[i])
                    sub_kind.append(kinds[i])
            node.child_node.append(create_decision_tree(sub_sa, sub_kind, attribute_set_co
py))

    # 选择了连续属性
    else:
        print("划分连续属性")
        divide_value_best = divide[best_a_index]
        node.divide_value = divide_value_best
        sub_sa_left = pd.DataFrame(data=None, columns=samples_set.columns)
        # print(sub_sa_left)
        # sub_kind_left = []
        sub_sa_right = pd.DataFrame(data=None, columns=samples_set.columns)
        # sub_kind_right = []

        # print(samples_set)
        count_left = 0
        count_right = 0
        # 将初始样本集和划分为左节点样本集合和右节点样本集合
        for i in range(len(samples_set)):
            # print()

            if samples_set.loc[i, best_attribute.name] < divide_value_best:
                count_left += 1
                sub_sa_left.loc[len(sub_sa_left)] = samples_set.loc[i, :]
            else:
                count_right += 1
                sub_sa_right.loc[len(sub_sa_right)] = samples_set.loc[i, :]
        # 左子树不为空，递归调用
        if len(sub_sa_left) != 0:
            print("jinxing_left_1")
            # print(sub_sa_left)
            nodet = create_decision_tree(sub_sa_left, attribute_set)
            # print(nodet.kinds)
            node.child_node.append(nodet)
        # 左子树为空，创建一个叶子节点
        else:
            print("jinxing_left_2")
            node_left = Node()
            node_left.kinds = node.kinds
            node.child_node.append(node_left)
        if len(sub_sa_right) != 0:
            print("jinxing_right_1")
            # print(sub_sa_right)
            nodet = create_decision_tree(sub_sa_right, attribute_set)
            # print(nodet.kinds)
            node.child_node.append(nodet)
        else:
            print("jinxing_ringth_2")
```

```
            node_right = Node()
            node_right.kinds = node.kinds
            node.child_node.append(node_right)

    return node
```

step3:决策树构建成功后，需要构建一个树的遍历函数。

使用这个函数预测样本的类别。

```
def dicide(node, sample):
    path = node.find_path(sample)
    # 寻找路径走不下去就停止，返回这个节点的类标
    if path == -1:
        return node.kinds
    else:
        return dicide(node.child_node[path], sample)
```

step4：进行决策树的评估。

通过上面创建的树和树的遍历函数，预测出测试样本集合中的每一个类别。

显示出预测值和实际值的差别。

然后构建混淆矩阵。

从而计算查准率和查全率以及 F1 的值

```
def evaluate(node, test_set):
    True_value = np.zeros(len(test_set)).tolist()
    Predict_value = np.zeros(len(test_set)).tolist()
    # 找到测试样本集合中每个样本的真实类标和预测类标
    for i in range(len(test_set)):
        Predict_value[i] = int(dicide(node, test_set.loc[i, :]))
        True_value[i] = int(test_set.loc[i, 'C'])

    print(type(True_value))
    # 生成一个混淆矩阵
    confusion_m = sklearn.metrics.confusion_matrix(True_value,Predict_value)

    # 输出评价指标
    print('c4.5决策树算法')
    print('真实值')
    print(True_value)
    print('预测值')
    print(Predict_value)
    print(type(confusion_m))
    print('**********************')
    print('测试数据集总数：', len(test_set))
    correct_num = confusion_m[0,0]+confusion_m[1,1]+confusion_m[2,2]
    print('正确率：',correct_num/len(test_set))
    print('\t\t混淆矩阵')
    print('真实情况\t1类\t2类\t3类')
    print(f'  1类\t{confusion_m[0,0]}   {confusion_m[0,1]}   {confusion_m[0,2]}')
    print(f'  2类\t{confusion_m[1,0]}   {confusion_m[1,1]}   {confusion_m[1,2]}')
    print(f'  3类\t{confusion_m[2,0]}   {confusion_m[2,1]}   {confusion_m[2,2]}')
    # confusion_m[:,2]
    P = np.zeros(3)
    R = np.zeros(3)
    F1 = np.zeros(3)
    for i in range(3):
        P[i] = confusion_m[i][i]/np.sum(confusion_m[:,i])
        R[i] = confusion_m[i][i]/np.sum(confusion_m[i,:])
        F1[i] = 2/(1/P[i]+1/R[i])

    print()
```

```
    for i in range(3):
        print(f'{i+1}类的查准率 = ',P[i])
        print(f'{i+1}类的查全率 = ',R[i])
        print(f'{i+1}类的F1 = ',F1[i])
    print('**********************')
```

step5：决策树的可视化显示。

将上面构建的决策树可视化。直观的感受树的样子。

总的来说，就是递归遍历构建的决策树，然后将每个节点应该显示的值表示好就行了。

```
def view_decision_tree(tree, dot, name,edge_label):
    global counter
    if len(tree.child_node) != 0 and tree.feature.type == 1:

        counter += 1
        node_name = 'node%d' % (counter)
        # node_label = tree.feature.name
        node_label = f'feature = {tree.feature.name}\n' \
                     f'kinds = {tree.kinds}\n' \
                     f'divide_value = {tree.divide_value:.3}\n'
        dot.node(name=node_name,label=node_label)
        if name!=None:
            dot.edge(tail_name=name, head_name=node_name, label=edge_label, color='red')

        left_edge_label = f'<={tree.divide_value:.3}'
        right_edge_label = f'>{tree.divide_value:.3}'
        view_decision_tree(tree.child_node[0],dot,node_name,edge_label=left_edge_label)
        view_decision_tree(tree.child_node[1], dot, node_name, edge_label=right_edge_label)


    # 叶子节点
    elif len(tree.child_node) == 0:
        counter += 1
        node_name = 'node%d' % (counter)
        node_label = f'kinds = {tree.kinds}\n'
        dot.node(name=node_name,label=node_label)
        dot.edge(tail_name=name, head_name=node_name, label=edge_label, color='red')
    return None


def writer(tree,filename):
    dot = Digraph(name="pic", comment="测试", format="png")
    dot.graph_attr['dpi'] = '500'
    view_decision_tree(tree=tree, dot=dot, name=None, edge_label=None)
    # print(dot)
    dot.render(filename=filename,
               directory='.',  # 当前目录
               view=True)
```

结果：

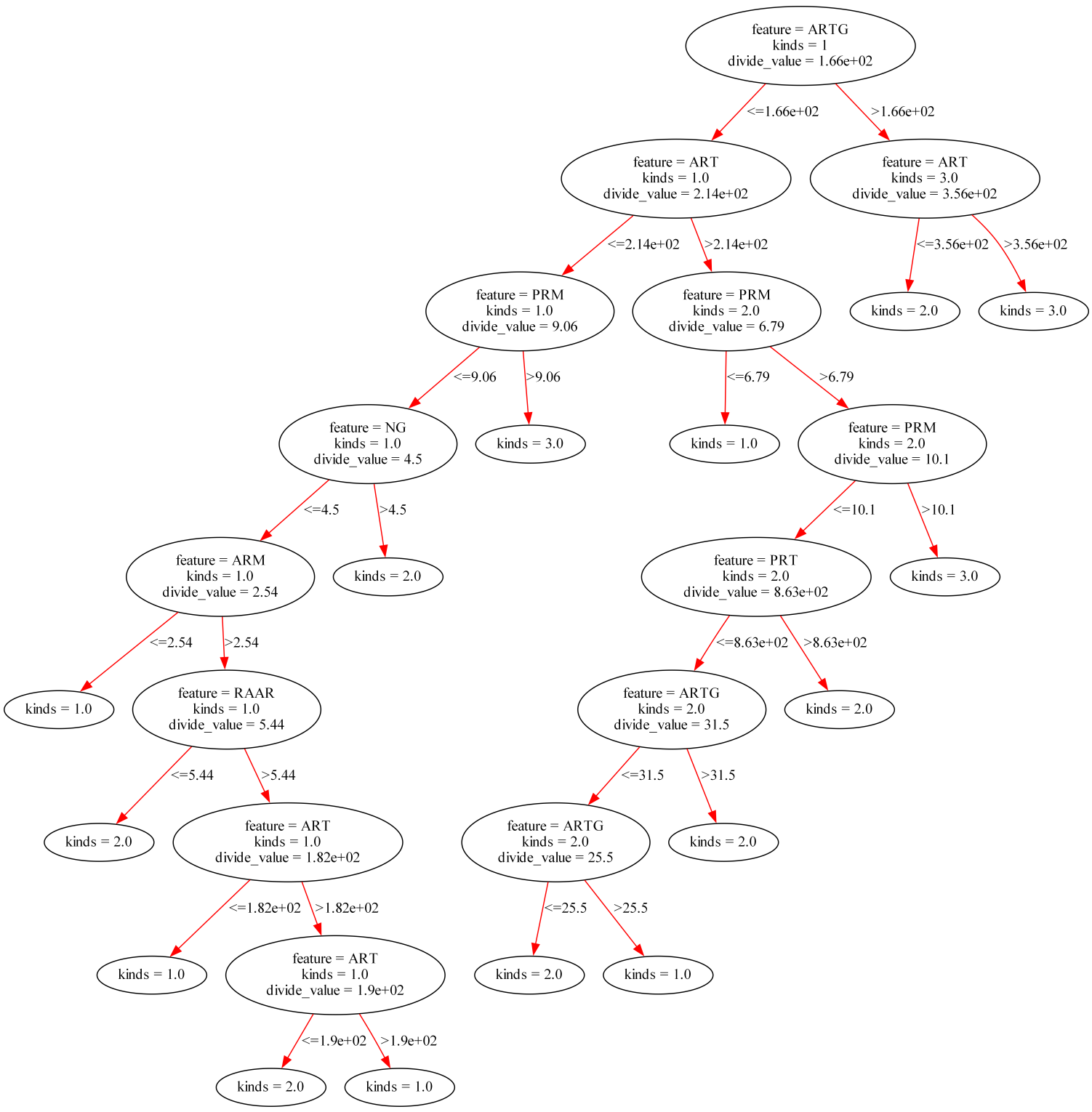
```
划分选择 [368.     79.5   608.      4.135   9.32  166.5     3.9    94.     39.98
   6.985]
选择的划分属性为： ARTG
[0.7235092174176074, 0.33415230213530095, 0.6153325947960536, 0.6323907899300543, 0.7656841096822061,
增益率为： 0.7689747140063712
连续属性的划分边界为： 166.5
这个节点的类别是： 1
划分连续属性
jinxing_left_1
划分选择 [214.5    65.    546.      2.545   7.03   47.      2.9    50.5    17.955
   2.78 ]
选择的划分属性为： ART
[0.5516264421680283, 0.2540926914078481, 0.4609714030378382, 0.47107857969961026, 0.4460918070145265,
增益率为： 0.5516264421680283
连续属性的划分边界为： 214.5
这个节点的类别是： 1.0
划分连续属性
```

图（3）进行划分属性选择的中间结果部分显示

调用截至同一类

| | C | ART | N | PRT | ARM | PRM | ARTG | NG | PRTG | RAAR | RAN |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.0 | 125.0 | 63.0 | 368.0 | 1.98 | 5.84 | 20.0 | 1.0 | 18.0 | 16.00 | 1.59 |
| 1 | 1.0 | 109.0 | 55.0 | 316.0 | 1.98 | 5.75 | 0.8 | 0.8 | 3.0 | 0.69 | 1.36 |
| 2 | 1.0 | 156.0 | 64.0 | 424.0 | 2.44 | 6.63 | 18.0 | 2.0 | 24.0 | 11.54 | 3.13 |
| 3 | 1.0 | 95.0 | 38.0 | 254.0 | 2.50 | 6.68 | 9.0 | 1.0 | 12.0 | 9.47 | 2.64 |
| 4 | 1.0 | 134.0 | 54.0 | 362.0 | 2.48 | 6.70 | 15.0 | 1.0 | 16.0 | 11.19 | 1.85 |
| 5 | 1.0 | 159.0 | 74.0 | 452.0 | 2.15 | 6.11 | 27.0 | 2.0 | 30.0 | 16.98 | 2.70 |
| 6 | 1.0 | 176.0 | 70.0 | 484.0 | 2.51 | 6.91 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |
| 7 | 1.0 | 141.0 | 56.0 | 366.0 | 2.52 | 6.54 | 9.0 | 1.0 | 12.0 | 6.38 | 1.79 |
| 8 | 1.0 | 142.0 | 45.0 | 328.0 | 2.52 | 6.54 | 9.0 | 1.0 | 12.0 | 6.38 | 1.79 |
| 9 | 1.0 | 200.0 | 91.0 | 576.0 | 2.20 | 6.33 | 27.0 | 2.0 | 30.0 | 13.50 | 2.20 |
| 10 | 1.0 | 114.0 | 61.0 | 346.0 | 1.87 | 5.67 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |
| 11 | 1.0 | 139.0 | 64.0 | 410.0 | 2.17 | 6.41 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |
| 12 | 1.0 | 139.0 | 64.0 | 410.0 | 2.17 | 6.41 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |
| 13 | 1.0 | 107.0 | 43.0 | 270.0 | 2.49 | 6.28 | 17.0 | 1.0 | 18.0 | 15.89 | 2.33 |
| 14 | 1.0 | 144.0 | 58.0 | 392.0 | 2.48 | 6.76 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |
| 15 | 1.0 | 153.0 | 64.0 | 434.0 | 2.39 | 6.78 | 42.8 | 3.8 | 49.0 | 27.94 | 5.86 |
| 16 | 1.0 | 135.0 | 59.0 | 390.0 | 2.29 | 6.61 | 12.0 | 1.0 | 14.0 | 8.89 | 1.70 |
| 17 | 1.0 | 55.0 | 23.0 | 146.0 | 2.39 | 6.35 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |
| 18 | 1.0 | 110.0 | 47.0 | 308.0 | 2.34 | 6.55 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |
| 19 | 1.0 | 187.0 | 76.0 | 492.0 | 2.46 | 6.47 | 33.0 | 3.0 | 40.0 | 17.65 | 3.95 |
| 20 | 1.0 | 125.0 | 52.0 | 348.0 | 2.40 | 6.69 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 |

图（4）：创建树的截至条件下的样本显示




图（5）：C4.5 决策树可视化结果

```
c4.5决策树算法
真实值
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
预测值
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 1, 2, 1, 2, 2, 2, 2, 1, 1, 1, 2, 1, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
<class 'numpy.ndarray'>
**********
测试数据集总数： 48
正确率： 0.8541666666666666
        混淆矩阵
真实情况  1类  2类  3类
  1类    15   1   0
  2类    6   10   0
  3类    0   0   16

1类的查准率 =  0.7142857142857143
1类的查全率 =  0.9375
1类的F1 =  0.8108108108108107
2类的查准率 =  0.9090909090909091
2类的查全率 =  0.625
2类的F1 =  0.7407407407407407
3类的查准率 =  1.0
3类的查全率 =  1.0
3类的F1 =  1.0
**********
```

图（6）C4.5 的评价指标和预测结果对比

## 问题 3：cart 决策树算法。

step1：构建 cart 决策树。

通过调用 python 机器学习库中的 DecisionTreeClassifier 函数创建一个 cart 决策树。

这个库可以设置树的深度。

```
1 def create_cart_tree(train_set):
2
```

```
    set = train_set.copy(deep=True)
    # print(set)
    y = set.loc[:, 'C']
    # print(y)
    set = set.drop(columns = 'C')
    X = set.loc[:,:]
    # print(X)
    # 构建树，限制树的深度为4
    tree = DecisionTreeClassifier(criterion='gini',max_depth=4).fit(X, y)
    dot_data = export_graphviz(tree, out_file=None,
                               filled=True, rounded=True,
                               special_characters=True,
                               precision=2)
    # 可视化决策树
    graph = pydotplus.graph_from_dot_data(dot_data)
    # im = Image(graph.create_png())
    graph.write_png('cart_tree.png')
    return tree
```

step2：进行 cart 树的评价。

使用 cart 树对测试集合中的样本进行预测，然后将预测值和实际值进行比较。

然后构建混淆矩阵。

从而计算查准率和查全率以及 F1 的值。

```
def evaluate_cart(tree,test_set):

    set = test_set.copy(deep=True)
    # print(set)
    y = set.loc[:, 'C'].tolist()
    y = np.array(y)
    y = y.astype(int)
    # print(y)
    set = set.drop(columns = 'C')
    X = set.loc[:,:]
    # print(X)
    # tree = DecisionTreeClassifier(max_depth=10, criterion='gini').fit(X, y)
    # tree = DecisionTreeClassifier( criterion='gini').fit(X, y)
    p_y = tree.predict(X)
    p_y.astype(int)
    # print(type(p_y))
    print('CART决策树算法')
    print('真实值')
    print(y)
    print('预测值')
    print(p_y)

    confusion_m = sklearn.metrics.confusion_matrix(y,p_y)
    print('**********************')

    print('测试数据集总数：', len(test_set))
    correct_num = confusion_m[0, 0] + confusion_m[1, 1] + confusion_m[2, 2]
    print('正确率：', correct_num / len(test_set))
    print('\t\t混淆矩阵')
    print('真实情况\t1类\t2类\t3类')
    print(f'  1类\t{confusion_m[0, 0]}   {confusion_m[0, 1]}   {confusion_m[0, 2]}')
    print(f'  2类\t{confusion_m[1, 0]}   {confusion_m[1, 1]}   {confusion_m[1, 2]}')
    print(f'  3类\t{confusion_m[2, 0]}   {confusion_m[2, 1]}   {confusion_m[2, 2]}')
    # confusion_m[:,2]
    P = np.zeros(3)
    R = np.zeros(3)
    F1 = np.zeros(3)
    for i in range(3):
        P[i] = confusion_m[i][i] / np.sum(confusion_m[:, i])
        R[i] = confusion_m[i][i] / np.sum(confusion_m[i, :])
```

```
        F1[i] = 2 / (1 / P[i] + 1 / R[i])

    print()
    for i in range(3):
        print(f'{i + 1}类的查准率 = ', P[i])
        print(f'{i + 1}类的查全率 = ', R[i])
        print(f'{i + 1}类的F1 = ', F1[i])
    print('**********************')

    return None
```

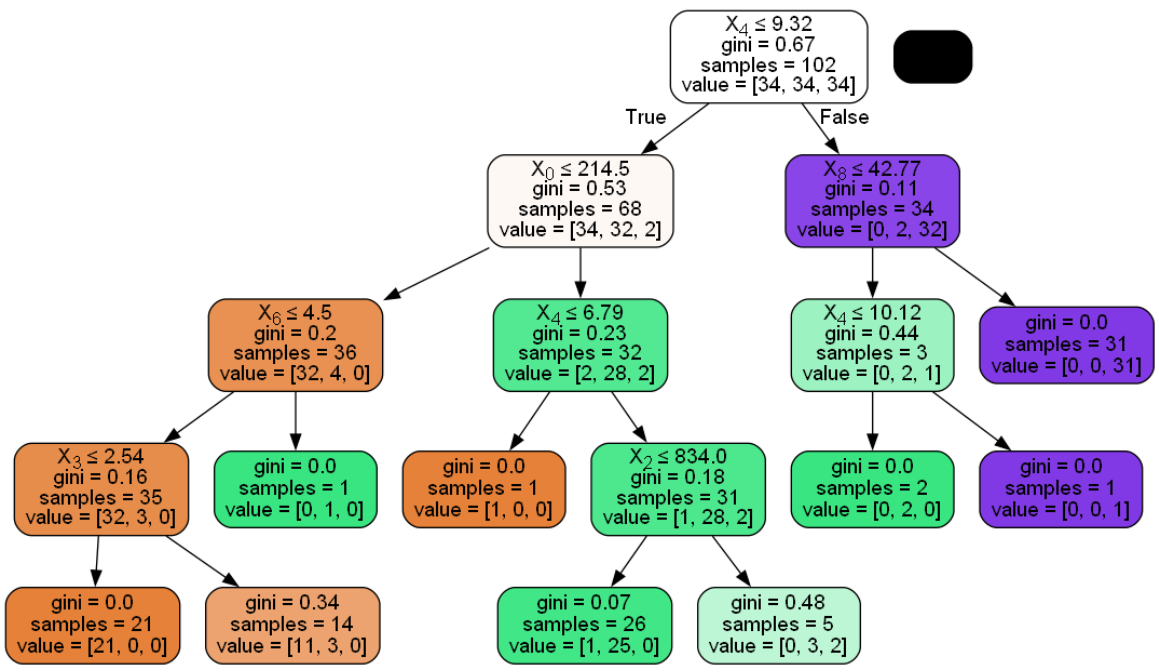
step3：进行两个树的比较

```
if __name__ == '__main__':
    samples_set, attribute_set = initial()
    train_set, test_set = train_and_test_divide(samples_set)
    print('训练集合')
    print(train_set)
    print('测试集合')
    print(test_set)
    tree = create_decision_tree(train_set, attribute_set)

    evaluate(tree, test_set)
    writer(tree,'c4.5_tree')

    cart_tree = create_cart_tree(train_set)
    evaluate_cart(cart_tree,test_set)
```

结果：

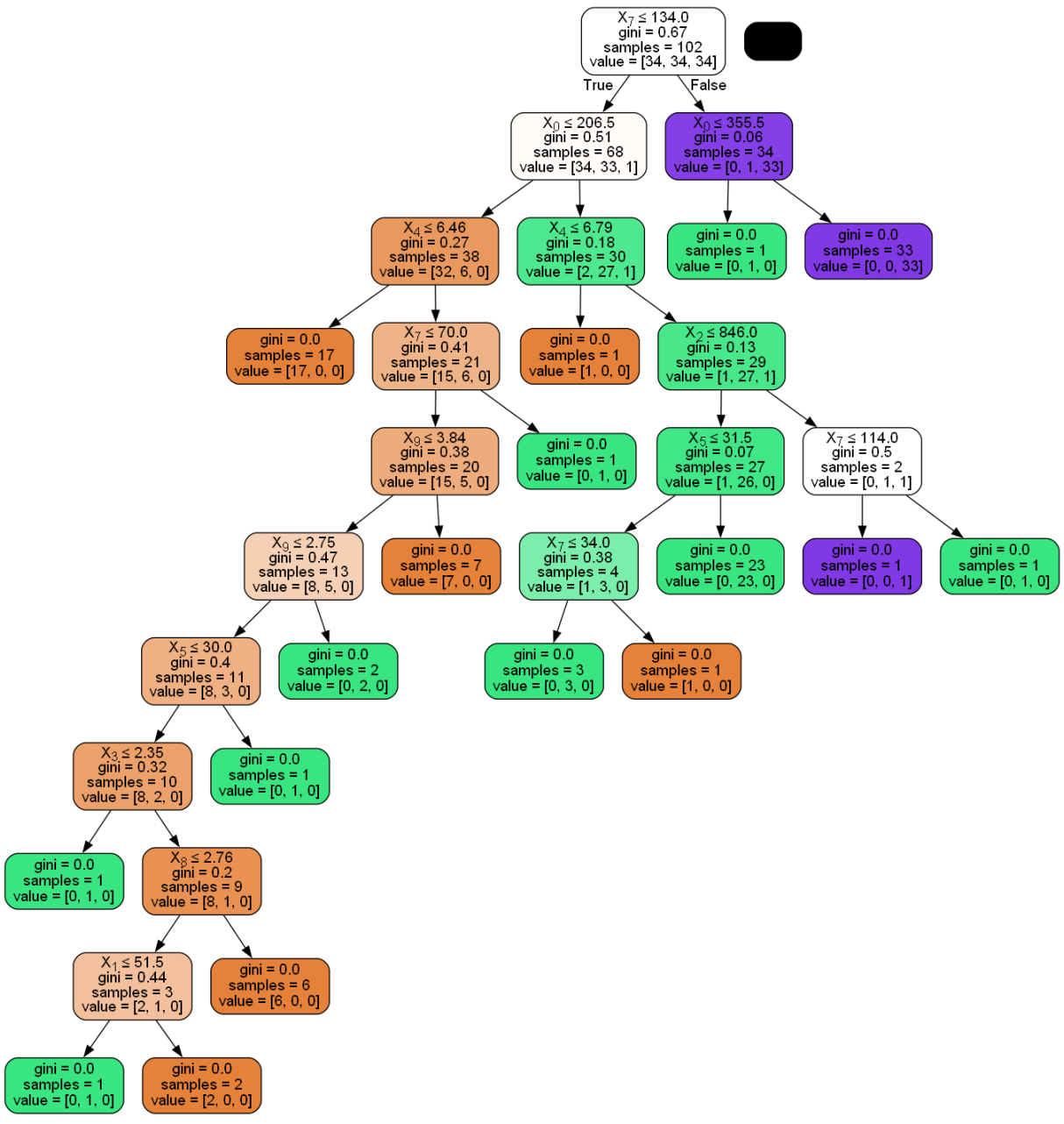



图（7）：CART 4 层深决策树

```
CART决策树算法
真实值
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
 3 3 3 3 3 3 3 3 3 3 3]
预测值
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 1 2 2 2 2 1 1 1 2 1 2 2 3 3 3 3 3
 2 3 3 3 3 3 3 3 3 3 3]
**********
测试数据集总数：  48
正确率：  0.875
          混淆矩阵
真实情况  1类  2类  3类
  1类    16   0   0
  2类    5   11   0
  3类    0   1   15

1类的查准率 =  0.7619047619047619
1类的查全率 =  1.0
1类的F1 =  0.8648648648648649
2类的查准率 =  0.9166666666666666
2类的查全率 =  0.6875
2类的F1 =  0.7857142857142856
3类的查准率 =  1.0
3类的查全率 =  0.9375
3类的F1 =  0.9677419354838711
**********
```

图（8）：CART 4 层决策树评价指标




图（9）：CART 不限制深度决策树

```
**********
测试数据集总数：  48
正确率：  0.7708333333333334
          混淆矩阵
真实情况  1类  2类  3类
  1类    13   3   0
  2类    4   10   2
  3类    1   1   14

1类的查准率 =  0.7222222222222222
1类的查全率 =  0.8125
1类的F1 =  0.7647058823529411
2类的查准率 =  0.7142857142857143
2类的查全率 =  0.625
2类的F1 =  0.6666666666666666
3类的查准率 =  0.875
3类的查全率 =  0.875
3类的F1 =  0.875
**********
```

图（10）：C4.5 评价

```
**********
测试数据集总数：  48
正确率：  0.7916666666666666
          混淆矩阵
真实情况  1类  2类  3类
  1类    12   4   0
  2类    3   11   2
  3类    0   1   15

1类的查准率 =  0.8
1类的查全率 =  0.75
1类的F1 =  0.7741935483870969
2类的查准率 =  0.6875
2类的查全率 =  0.6875
2类的F1 =  0.6875
3类的查准率 =  0.8823529411764706
3类的查全率 =  0.9375
3类的F1 =  0.9090909090909091
**********
```

图（11）：CART 评价

进行50次构建和分类得到的平均指标

| | **类1** | | **类2** | | **类3** | |
|---|---|---|---|---|---|---|
| | C4.5 | cart | C4.5 | cart | C4.5 | cart |
| P | 0.79 | 0.82 | 0.72 | 0.80 | 0.88 | 0.89 |
| R | 0.79 | 0.87 | 0.66 | 0.70 | 0.92 | 0.94 |
| F1 | 0.78 | 0.84 | 0.68 | 0.74 | 0.90 | 0.91 |

通过上面实验发现一下特点：

1.CART 决策树的分类准确性一般高于 C4.5 决策树。

2.CART 如果限制树的深度可以得到更好的树，这可能是由于提高了树的泛化能力，从而使得在测试集合中的表现更好。

3.不同的训练集合和测试集的划分得出来的决策树性能差距可能很大，所有好的样本集合划分也可以提高决策树的性能。

## 三、 总结（心得体会）

这次实验主语难点是要自己实现 C4.5 决策树的算法。

这个算法实现中有很多细节问题需要注意，根据书中的伪代码实现时，要注意一下细节实现问题，比如对连续属性值进行最优划分点的选择时，需要注意排除掉重复值，如果有多个重复值存在，使用中位数作为边界值是可能会形成同样的边界值，这样会导致死循环。还有选择最优划分边界时，一定要使用信息增益来进行划分，不然会导致树的性能太低。