西北大学

智能信息系统综合实践
实验报告

题　　目：　决策樹

年　　级：　2020 级

专　　业：　软件工程

姓　　名：　牛强

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

首先定义一个树结点类

```
class Node:
    def __init__(self, feature=None, pivot=None, label=None, left=None, right=None, depth=None):
        self.feature = feature  # 分割特征
        self.pivot = pivot  # 分割特征值
        self.label = label  # 叶子节点的标签
        self.left = left
        self.right = right
        self.depth = depth  # 深度
```

然后定义计算熵函数，通过 np.unique()返回数组中不同元素种类个数，通过公式 $E=-\sum_{i=1}^{n} p(x_i)\log p(x_i)$ 获得熵值

```
def entropy(y):
    _, counts = np.unique(y, return_counts=True)  # 得到其数量
    p = counts / len(y)
    return -np.sum(p * np.log2(p))
```

接下来是计算信息增益，其中 X 为训练集，y 为对应标签，feature 为特征。

首先将训练集的某一特征的所有值转换为数组并进行排序

```
def information_gain(X, y, feature):
    n = X[feature]
    n = n.tolist()
    n = set(n)  # 去重
    n = list(n)  # 转为列表
    n = np.sort(n)
```

因为有连续值的出现，所以需要采用对应的处理方法。假设共有 n 个值，将某一特征连续值按照大小排序后，每两个相邻的数值取中位数作为划分界限，然后计算其对应的信息增益，共取 n-1 次，最终保留信息增益最大值。

函数返回信息增益，划分值和划分位置，以便后续划分操作。

```
ig = 0
p = 0
pivot_idx = 0
for i in range(len(n) - 1):
    # 求中位数值
    pivot = (n[i] + n[i+1]) / 2.0
    left_idx = X[feature] < pivot  # 赋予布尔值
    right_idx = X[feature] >= pivot
    left_y, right_y = y.loc[left_idx], y.loc[right_idx]  # 提取行数据
    e1, e2 = entropy(left_y), entropy(right_y)
    w1, w2 = len(left_y) / len(y), len(right_y) / len(y)
    ig1 = entropy(y) - w1 * e1 - w2 * e2
    if ig1 > ig:
        pivot_idx = i
        p = pivot
        ig = ig1
return ig, p, pivot_idx
```

然后定义构建决策树函数，其中 X 为训练集，y 为对应标签，feature 为特征，depth 为当前深度。

```
def build_tree(X, y, features, depth):
```

首先是退出条件

1. 对深度进行简单的控制，防止过拟合
2. 如果所有数据都属于同一类别，则返回叶子节点
3. 如果没有特征可用，则返回叶子节点，该叶子节点标签为出现最多的类别。

```
# 控制深度，返回叶子节点，该叶子节点标签为出现最多的类别
if depth > 7:
    label = y.value_counts().idxmax()
    return Node(label=label)

# 如果所有数据都属于同一类别，则返回叶子节点
if len(np.unique(y)) == 1:
    return Node(label=y.iloc[0])

# 如果没有特征可用，则返回叶子节点，该叶子节点标签为出现最多的类别
if len(features) == 0:
    label = y.value_counts().idxmax()
    return Node(label=label)
```

接下来计算信息增益并选取大于平均值的属性

```
# 计算ig并选取大于平均值的
ig = np.zeros(len(features))
pivot = np.zeros(len(features))
pivot_idx = np.zeros(len(features))
i = 0
for feature in features:
    ig[i], pivot[i], pivot_idx[i] = information_gain(X, y, feature)
    i = i+1
# 选取大于平均信息增益值的属性
feature_idx = ig >= np.median(ig)   # true or false
```

然后计算信息增益率，首先初始化一些变量。

```
# 计算信息增益率
i = 0
max_gain_ratio = 0
best_feature = features[i]
pivot_value = 0
```

接着对那些信息增益大于平均值的属性计算信息增益率，并选取最大者，更新最佳属性及其划分值，接着对训练集进行划分。

```
for feature in features:
    if feature_idx[i] == False:
        i = i + 1
        continue
    num = len(X)
    x = pivot_idx[i] + 1  # 左侧数据数量
    z = num - x  # 右侧数据数量
    iv = -x/num * np.log2(x/num) - z/num * np.log2(z/num)
    gain_ratio = ig[i] / iv
    if gain_ratio > max_gain_ratio:
        # 更新max和特征及其对应的划分值pivot下标
        max_gain_ratio = gain_ratio
        best_feature = features[i]
        pivot_value = pivot[i]
    i = i + 1
```

```
# 划分训练集
left_idx = X[best_feature] < pivot_value
right_idx = X[best_feature] >= pivot_value
left_X = X.loc[left_idx]
left_y = y.loc[left_idx]
right_X = X.loc[right_idx]
right_y = y.loc[right_idx]
```

在书中还有一种情况是“如果划分集合为空，则返回叶子节点，该叶子节点标签为出现最多的类别”，但在连续值中并不会出现这样的现象，因为每次选取的是两个数的中位数，左右两侧的划分集合一定会至少有一个元素，而当元素只有一个时将满足退出条件 2，即“如果所有数据都属于同一类别，则返回叶子节点”。

最后去除已选择的特征，进行递归建树

```
# 除去已选择特征
features = [f for f in features if f != best_feature]
# 递归建树
left = build_tree(left_X, left_y, features, depth+1)
right = build_tree(right_X, right_y, features, depth+1)

return Node(feature=best_feature, pivot=pivot_value, left=left, right=right, depth=depth+1)
```

接着还需要定义预测函数，其中 tree 为树结点，X 为测试数据。通过比较 X 的结点属性值与划分值的大小，然后进行自身调用，直到叶子节点。

```
def predict(tree, X):
    if tree.label is not None:
        return tree.label
    if X[tree.feature] < tree.pivot:
        return predict(tree.left, X)
    else:
        return predict(tree.right, X)
```

然后简单绘制出决策树图像，首先要获得树的左右大小与深度。

```
def get_tree_size(node):
    if node is None:
        return 0, 0
    left_width, left_height = get_tree_size(node.left)
    right_width, right_height = get_tree_size(node.right)
    return left_width + right_width + 1, max(left_height, right_height) + 1
```

再通过 annotate 绘制结点，plot 绘制线条。

```
def plot_tree_node(ax, x, y, node, parent_x=None, parent_y=None):
    if node.feature is not None:
        s1 = 'fea:'+str(node.feature)+'\n'+\
             'p:'+str(np.round(node.pivot, 2))
    else:
        s1 = 'l:'+str(node.label)

    ax.annotate(s1, xy=(x, y), xycoords='data',
                xytext=(0, 10), textcoords='offset points',
                ha='center', va='bottom')
    if parent_x is not None and parent_y is not None:
            ax.plot([parent_x, x], [parent_y, y], color='red')
```

使用以下函数对绘制函数进行调用，通过 h 和 v 来调整图像

```
def plot_tree(ax, node, x=0, y=0, parent_x=None, parent_y=None):
    if node is None:
        return
    plot_tree_node(ax, x, y, node, parent_x, parent_y)
    h_spacing = 3
    v_spacing = 8
    left_width, left_height = get_tree_size(node.left)
    right_width, right_height = get_tree_size(node.right)
    plot_tree(ax, node.left, x - h_spacing * (left_width + right_width + 1) // 2, y - v_spacing, x, y)
    plot_tree(ax, node.right, x + h_spacing * (left_width + right_width + 1) // 2, y - v_spacing, x, y)
```

最后使用一个函数当作树结点入口

```
def plot_binary_tree(root):
    fig, ax = plt.subplots(figsize=(10, 5))
    ax.axis('off')
    ax.set_aspect('equal')
    width, height = get_tree_size(root)
    plot_tree(ax, root, width / 2, height - 1)
    plt.show()
```

接下来进行主函数部分

首先读取数据，软木塞数据已经以 xlsx 的形式存放在根目录下。

```
# 读取数据
data = pd.read_excel('data.xlsx')
```

然后去除一些无用值，并对特征和标签进行分离，便于函数处理。

```
# 分离特征和标签
X = data.drop('C', axis=1)
X = X.drop('#', axis=1)
y = data['C']
```

使用 train_test_split 方法随机划分训练集和测试集

```
# 随机划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

接着建立树模型，其中 X_train.columns 是特征集合

```
# 建立C4.5决策树模型
tree1 = build_tree(X_train, y_train, X_train.columns, depth=1)
```

通过 apply(lambda x:)方法调用 predict 函数进行预测

```
# 对测试数据集进行预测
y_pred = X_test.apply(lambda x: predict(tree1, x), axis=1)
```

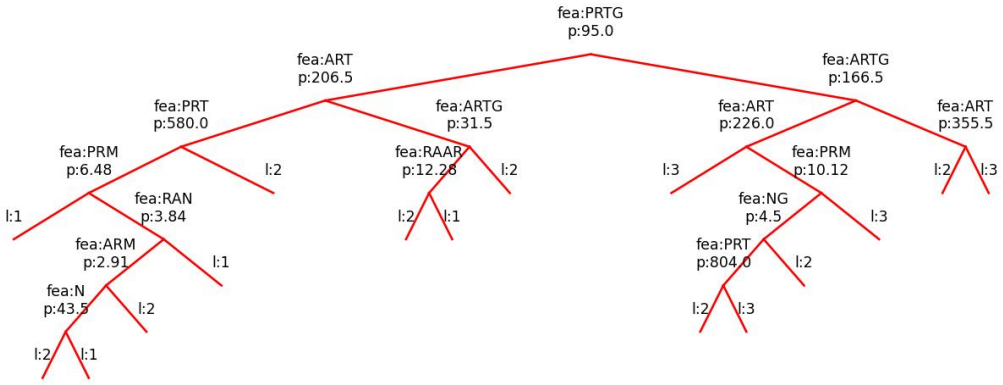
调用 sklearn.metrics 中的方法对决策树进行评估

```
# 计算混淆矩阵
cm = confusion_matrix(y_test, y_pred)

# 计算查准率、查全率和F1值
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')
f1 = f1_score(y_test, y_pred, average='macro')

# 打印结果
print("C4.5 Confusion Matrix:\n", cm)
print("Precision: ", precision)
print("Recall: ", recall)
print("F1 Score: ", f1)
```

其运行结果为：

```
C4.5 Confusion Matrix:
 [[ 7  3  0]
 [ 2  4  0]
 [ 0  0 14]]
Precision:  0.7830687830687831
Recall:  0.7888888888888889
F1 Score:  0.7840755735492576
```

最后绘制出决策树图像



考虑到软木塞数据集属性都是连续值，经过连续值处理后等同于每个属性只有两种可取值，即小于中位数和大于中位数。而 C4.5 算法是为了防止增益对取值数目较多的属性有所偏好，那么对于这道题来说是否可以不使用 C4.5 中的信息增益率而只是对 ID3 算法改进，进行连续值处理？

以下是 C4.5 与 ID3（进行连续值处理）决策树对软木塞数据集训练并测试的评估结果（P、R、F1），每次两种决策树使用相同的训练集与测试集，进行了多次测试取平均值。

表 1 C4.5 与 ID3 决策树对软木塞数据集训练评估结果

C4.5			ID3		
P	R	F1	P	R	F1
0.78307	0.78889	0.78408	0.78307	0.78889	0.78408
0.82336	0.81818	0.81667	0.97222	0.96970	0.96963
0.78315	0.80000	0.76769	0.85641	0.86111	0.83177
0.76667	0.74881	0.75556	0.76667	0.74881	0.75556
0.82183	0.83333	0.81263	0.84392	0.86111	0.84371
0.89773	0.89815	0.89335	0.89773	0.89815	0.89335
0.69451	0.72222	0.70503	0.80556	0.81852	0.80335
0.68783	0.70000	0.68025	0.58532	0.60000	0.57386
0.90741	0.90303	0.90389	0.88141	0.86970	0.87037
0.71937	0.72487	0.72099	0.71937	0.72487	0.72099
0.83929	0.80842	0.80934	0.83929	0.80842	0.80934
0.84982	0.83636	0.82548	0.76768	0.76229	0.76132
0.77289	0.77289	0.72525	0.84848	0.87179	0.83069
0.73553	0.75758	0.73148	0.73553	0.75758	0.73148
0.85185	0.89412	0.86865	0.86806	0.89412	0.87879
0.81250	0.79720	0.79673	0.65079	0.66472	0.65394
0.58498	0.59848	0.58889	0.60995	0.62879	0.61544
0.84259	0.84848	0.84126	0.80168	0.80682	0.80265
0.81861	0.80952	0.79535	0.72937	0.72619	0.72663
0.87179	0.86111	0.83292	0.83217	0.83333	0.79278
0.71591	0.71944	0.71679	0.71591	0.71944	0.71679
0.78307	0.78319	0.77460	0.80879	0.81097	0.80592
0.85185	0.70527	0.71197	0.75694	0.70527	0.71197
0.85185	0.85185	0.84127	0.79084	0.79630	0.76874

其中灰色标注为两种决策树评估结果相同情况，偶尔也会出现树结构不同，但 P 值相同的情况，应该是与训练集、测试集划分有关。

对表 1 数据进行均值计算得到表 2，总体来说两种决策树模型的预测结果差距不是很明显。

表 2 C4.5 与 ID3 决策树评估均值

	P_Average	R_Average	F1_Average
C4.5	0.79448	0.79089	0.77917
ID3	0.78613	0.78862	0.77721

接下来是使用 CART 算法构造决策树模型

```
# 训练CART分类树
model = DecisionTreeClassifier(criterion='gini', splitter='best',
                               max_depth=7, min_samples_split=2,
                               min_samples_leaf=1, max_features=None,
                               class_weight=None)
model.fit(X_train2, y_train2)
```

criterion：用于衡量节点纯度的度量方式。

splitter：用于选择每个节点的拆分策略。'best'表示选择最优的拆分方式

max_depth：树的最大深度。

min_samples_split：一个节点在拆分前必须具有的最小样本数。默认值是 2，表示每个节点至少有 2 个样本才能进行拆分。

min_samples_leaf：叶节点的最小样本数。默认值是 1，表示每个叶节点至少有 1 个样本。

```
# 在测试集上进行预测
y_pred2 = model.predict(X_test2)
```

```
# 计算混淆矩阵、查全率、查准率和F1值
cm3 = confusion_matrix(y_test2, y_pred2)
recall3 = recall_score(y_test2, y_pred2, average='macro')
precision3 = precision_score(y_test2, y_pred2, average='macro')
f3 = f1_score(y_test2, y_pred2, average='macro')

print('CART Confusion matrix:\n', cm3)
print('Precision: %.3f' % precision3)
print('Recall: %.3f' % recall3)
print('F1: %.3f' % f3)
```

输出结果如下：

```
CART Confusion matrix:
 [[12  2  0]
 [ 1  3  1]
 [ 0  2  9]]
Precision: 0.751
Recall: 0.758
F1: 0.749
```

最后生成决策树图像

```
# 生成决策树图形
dot_data = export_graphviz(model, out_file=None, feature_names=X_train.columns,
                           class_names=['1', '2', '3'], filled=True,
                           rounded=True, special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("cork_decision_tree")

# 显示决策树图形
graph.view()
```

model: 要导出的决策树模型对象。

out_file: 导出文件路径。如果为 None，则返回生成的 Graphviz 格式字符串。

feature_names: 特征名称列表，表示决策树上各节点所代表的特征。

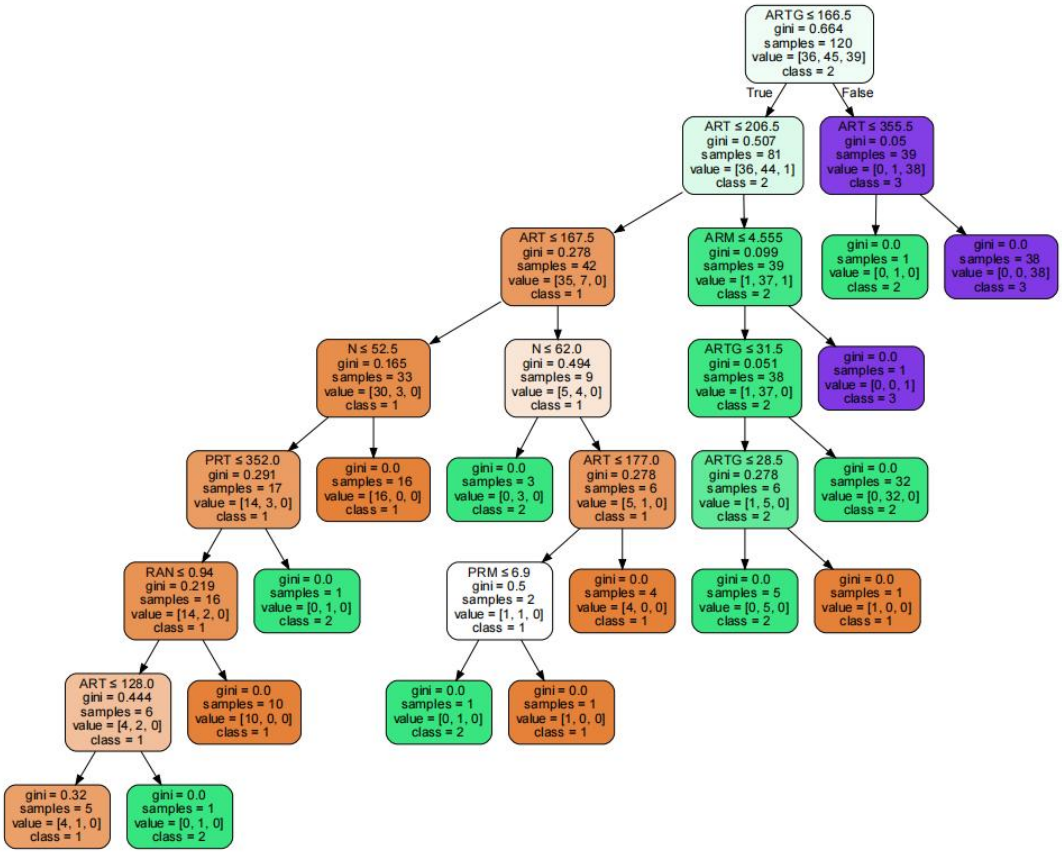
class_names: 类别名称列表，表示决策树分类的各个类别。

filled: 是否给树节点填充颜色，颜色深度表示该节点分类的置信度。

rounded: 是否将树节点框圆角化。

special_characters: 是否允许特殊字符的显示。

生成图像如下：



接着使用同样的训练集与测试集，分别构建 C4.5 决策树模型和 CART 决策树模型，并对其进行 50 次训练与评估，对评估结果取均值。通过结果可知两者相差并不是很大，性能不分伯仲。

表 3 C4.5 与 CART 评估均值

	P_Average	R_Average	F1_Average
C4.5	0.81435	0.81299	0.80643
CART	0.80747	0.80042	0.79435

表 4 C4.5 与 CART 决策树对软木塞数据集训练评估结果

C4.5			CART		
P	R	F1	P	R	F1
0.7619	0.7357	0.7280	0.8330	0.8060	0.7960
0.8977	0.8956	0.8950	0.8330	0.8290	0.8290
0.8259	0.8241	0.8237	0.8670	0.8610	0.8600
0.8630	0.8674	0.8593	0.8630	0.8670	0.8590
0.8558	0.8455	0.8398	0.8150	0.8120	0.8090
0.9296	0.9389	0.9326	0.9020	0.9110	0.9020
0.9091	0.9000	0.8885	0.8580	0.8580	0.8520
0.7611	0.7744	0.7592	0.8060	0.7980	0.7950
0.7765	0.7778	0.7731	0.7500	0.7410	0.7390
0.8487	0.8259	0.8257	0.8160	0.7850	0.7870
0.7355	0.7354	0.7337	0.7720	0.7610	0.7650
0.8424	0.8500	0.8360	0.8140	0.8000	0.7690

CART 与 C4.5 算法优缺点

C4.5 优点：

1. C4.5 是 ID3 的改进，采用信息增益率进行特征选择，能够有效地处理低维度和高维度数据集。

2. 决策树模型具有很好的可解释性，能够为决策提供直观的解释。

缺点：

1. 决策树模型复杂度较高，容易过拟合。

2. 计算复杂度较高，需要耗费较长的训练时间。

CART 优点：

1. 可用来解决分类问题和回归问题。

2. 生成的决策树模型是二叉树，简单、易于理解和实现。

缺点：

1. 基尼指数偏向于处理高维度数据和特征取值较多的数据集。

2. CART 算法生成的是二叉决策树，因此可能会导致一些信息的损失。

3. 对于噪声数据和异常值较为敏感。

三、总结

根据实验结果，对于软木塞数据集这种属性都是连续值的情况，我认为可以不使用 C4.5 算法中的信息增益率，而是只对 ID3 算法进行连续值处理。

ID3 算法在处理连续值特征时，一般会将每个特征的取值按照大小排序，然后选取相邻两个取值的中点作为划分点。这样，每个连续值特征就可以转化为一个离散值特征，即小于等于划分点和大于划分点两种取值。这种方法虽然有一定的信息损失，但在实际应用中通常能够得到比较好的结果。当然，具体算法选择还应该根据数据集的特点和具体任务进行综合考虑。

CART 与 C4.5 预测结果评估相差很小，性能不分伯仲。