《机器学习》编程作业 2

题目 1.1，2.1，2.2，2.3

一、题目理解

1.1 题目 1.1

实现基于信息增益(ID3)和信息增益率(C4.5)的决策树算法，并为表 4.3 中数据生成一颗决策树，并做性能评价。

理解：建立决策树算法，分别采用信息增益（ID3）以及信息增益率（C4.5）的划分方法，通过不同的实验条件设置（将西瓜数据集 3.0 划分为测试集与训练集，可以设置不同的划分比例），进行多次训练，从而得到不同的决策树生成结果，针对条件与结果进行性能评价。

1.2 题目 2.1

复用[1]的算法，在第三章选择的 UCI 数据集上生成一颗决策树，并比较未剪枝、预剪枝、后剪枝策略的性能评价。

理解：选用信息增益的划分方式，数据集选用 UCI 数据集乳腺癌数据集，划分训练集以及测试集，分别采用不剪枝、预剪枝以及后剪枝的剪枝方式进行训练，得到不同的训练结果。

1.3 题目 2.2

在第三章选择的 UCI 数据集上，实现基于基尼指数划分选择(CART）的决策树算法，并比较未剪枝、预剪枝、后剪枝策略的性能评价。

理解：在题目 2.1 的基础上，将划分方式改变成基尼指数，同 2.1 的操作。

1.4 题目 2.3

以参数 MaxDepth 控制树的最大深度，设计不使用递归的决策树生成算法 C4.5。

理解：不采用递归生成树，引入队列 queue，在创建树结点的过程中将结点按照广度优先的顺序存入队列，并按顺序取出处理，为每个结点生成子节点（除叶节点外）。构建一个辅助队列，存入所有节点，在遍历完 queue1 后，利用辅助队列更新所有节点的 leaf_num（以当前节点作为根节点，该子树的叶子结点数量）。

二、算法原理阐述

2.1 决策树算法原理

决策树是一种常见的分类与回归方法，采用树结构来表示决策过程。决策树由根节点、内部节点和叶节点组成。每个非叶节点代表一个特征属性的测试，每个分支则对应于该特征

属性在某个值域上的输出，而每个叶节点存储一个类别或数值。

1. 树结构概述

根节点：代表所有样本的起始点。
内部节点：表示对特征属性的测试。
叶节点：表示决策结果或输出值。

从根节点到每个叶节点的路径代表了一个决策的过程，每一步的决策都是基于当前节点的特征属性进行的。最终到达的叶节点就是分类结果或预测值。

2. 决策树构建过程

构建决策树的核心在于选择最优特征进行划分，常用的选择标准包括信息增益、信息增益比和基尼指数等，这三种选择标准的算法可参见下文的原理介绍。

决策树学习的基本算法如下：

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
　　属性集 $A = \{a_1, a_2, \ldots, a_d\}$.
过程： 函数 TreeGenerate(D, A)
1: 生成结点 node;
2: **if** D 中样本全属于同一类别 C **then**
3: 　将 node 标记为 C 类叶结点; **return**
4: **end if**
5: **if** $A = \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
6: 　将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; **return**
7: **end if**
8: 从 A 中选择最优划分属性 a_*;
9: **for** a_* 的每一个值 a_*^v **do**
10: 　为 node 生成一个分支; 令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;
11: 　**if** D_v 为空 **then**
12: 　　将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; **return**
13: 　**else**
14: 　　以 TreeGenerate(D_v, $A \setminus \{a_*\}$)为分支结点
15: 　**end if**
16: **end for**
输出： 以 node 为根结点的一棵决策树

图 1 决策树学习基本算法

决策树的生成是一个递归过程，其递归的终止条件一般有：

（1）所有样本属于同一类别：如果当前节点的所有样本都属于同一类别，则该节点成为叶子节点，并标记为该类别。

（2）没有特征可供选择：如果当前节点没有特征可供选择（所有特征都已使用），则该节点成为叶子节点，并标记为当前节点中样本数最多的类别。

（3）达到预设的深度：如果决策树的深度达到了预设的最大深度，则停止递归。

（4）样本数过少：如果当前节点的样本数少于某个预设的阈值，则停止递归。

2.2 信息增益（ID3）

信息增益（Information Gain）是决策树算法中用于选择特征的一种标准，特别是在 ID3（Iterative Dichotomiser 3）算法中。信息增益的原理基于信息论中的熵（Entropy）概念，用于衡量数据集的不纯度或混乱程度。

1. 熵（Entropy）

熵（Entropy）是用于衡量信息不确定性的度量。对于数据集 D，其熵 H(D)定义为：

$$H(D) = -\sum_{j=1}^{c} p_j \log_2(p_j)$$

其中，p_j是类别 j 的概率，c 是类别数。熵越大，数据集的不确定性越高。

2. 信息增益的计算

信息增益衡量通过特征 A 划分数据集后，不确定性减少的程度。设数据集 D 经过特征 A 划分为子集 D1,D2,…,Dk,则信息增益 IG(D,A)的计算公式为：

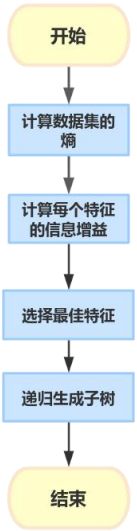
$$IG(D, A) = H(D) - \sum_{i=1}^{k} \frac{|D_i|}{|D|} H(D_i)$$

这里，｜Di｜是子集 Di 的样本数量，｜D｜是数据集 D 的样本总数。信息增益越大，表示使用特征 A 划分数据集后的不确定性减少越显著。

3. 特征选择

ID3 算法通过计算所有特征的信息增益，选择信息增益最大的特征作为当前节点的划分特征。该过程会递归进行，直到满足停止条件（如所有样本属于同一类别或没有可用特征）。

4. ID3 算法步骤



计算数据集的熵：首先计算整个数据集 D 的熵 H(D)。

计算每个特征的信息增益：对于每个特征 A，计算其条件熵 H(D｜A)，然后计算信息增益 IG(D,A)。

选择最佳特征：选择信息增益最大的特征作为当前节点的划分特征。

递归生成子树：根据选定的特征，将数据集划分为若干个子集，对每个子集递归地执行上述步骤，直到满足终止条件。

2.3 信息增益率（C4.5）

C4.5 算法是对 ID3 算法的改进，主要通过引入信息增益率来选择特征，从而解决 ID3 算法中信息增益偏向于选择多值特征的问题。

1. 信息增益率的定义

信息增益率（Gain Ratio）是在计算信息增益的基础上，引入特征的固有信息（Intrinsic Information），以减少多值特征对特征选择的偏见。信息增益率的计算公式为：

$$Gain\ Ratio(D,A) = \frac{IG(D,A)}{IV(A)}$$

其中，IV(A)表示特征 A 的固有信息，定义为：

$$IV(A) = -\sum_{i=1}^{k} \frac{|D_i|}{|D|} \log_2 \left(\frac{|D_i|}{|D|} \right)$$

这里，Di 是通过特征 A 划分得到的子集。信息增益率越大，表示该特征的选择越优。

2. 特征选择与构建树

C4.5 算法的特征选择步骤与 ID3 类似，但它选择信息增益率最大的特征作为划分特征。算法的实现步骤如下：

1. 计算每个特征的信息增益和固有信息。
2. 计算每个特征的信息增益率。
3. 选择信息增益率最大的特征进行划分。
4. 对每个子集递归执行上述步骤，构建决策树。
5. 执行剪枝以提高模型的泛化能力。

2.4 基尼系数

在机器学习中，基尼系数常用于决策树算法，尤其是 CART（Classification and Regression Trees）算法中，作为特征选择的标准。

1. 基尼不纯度

基尼系数被用来衡量一个数据集的纯度。在分类问题中，基尼不纯度定义为：

$$Gini(D) = 1 - \sum_{i=1}^{C} p_i^2$$

其中，pi 是数据集中属于类别 i 的样本比例，C 是类别的总数。基尼不纯度越小，表示数据集越纯，即同一类别的样本越集中。

2. 特征选择

在构建决策树时，算法会评估每个特征的基尼不纯度，以决定如何划分数据。选择使得子集基尼不纯度最小的特征进行分裂，从而提高模型的预测准确性。

3. 划分数据集

假设特征 X 将数据集 D 划分为两个子集 D_1和 D_2，则划分的基尼不纯度计算为：

$$Gini_{split} = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2)$$

其中，|D| 是数据集中样本的总数，$|D_1|$ 和 $|D_2|$ 是划分后子集的样本数。

4. 最优划分

通过计算所有特征的基尼不纯度，算法选择基尼不纯度最小的特征进行划分，反复进行，直到达到停止条件（如树的深度限制或节点样本数小于某个阈值）。

2.5 预剪枝

预剪枝（Pre-pruning）是一种防止决策树过拟合的技术，通过在树的构建过程中限制树的生长来实现。

1. 原理

预剪枝的核心思想是在构建决策树时，通过设置条件来决定是否继续对当前节点进行划分。预剪枝的目标是防止决策树变得过于复杂，从而提高模型的泛化能力。具体来说，每次划分时都会评估划分后的效果，如果划分后的模型在验证集上的性能没有显著提升，则不再继续划分。

2. 实现方式

（1）选择停止条件：
节点样本数阈值：设置一个最小样本数阈值，当节点样本数低于该阈值时，不再进行划分。
性能评估：在划分后计算模型在验证集上的性能（如准确率、F1 分数等），如果性能未提升，则停止划分。
复杂度限制：可设定最大树深度或最大叶节点数，超过限制则停止划分。
（2）构建决策树：
从根节点开始，逐层测试每个特征进行划分。
每次划分时，先计算当前节点的性能。
若满足停止条件，则将当前节点设为叶节点，存储类别；否则继续划分。
（3）性能监控：在每次划分后，使用验证集监控模型的性能变化。

2.6 后剪枝

后剪枝（Post-pruning）是一种在决策树构建完成后，通过评估子树对模型性能的贡献来减少树的复杂度的技术。

1. 原理

后剪枝的核心思想是在构建出完整的决策树后，检查每个子树是否能被剪除，以提高模型的泛化能力。通过去除那些在验证集上表现不佳的子树，后剪枝可以有效减少过拟合现象。

2. 实现步骤

（1）构建完整的决策树：首先，利用训练集构建出完整的决策树。
（2）评估每个子树：从叶节点开始，逐层向上遍历树的每个子树。对于每个子树，评估其对模型性能的贡献，通常通过验证集来判断。
（3）计算性能指标：计算当前子树的预测准确率 P_{subtree}，然后计算合并后的节点（父节点）的预测准确率 P_{parent}。
（4）决策剪枝：如果合并后的准确率P_{parent}不低于子树的准确率P_{subtree}，则可以剪除该子树，合并为父节点。
（5）继续遍历：重复以上步骤，直到没有更多的子树可以剪除。

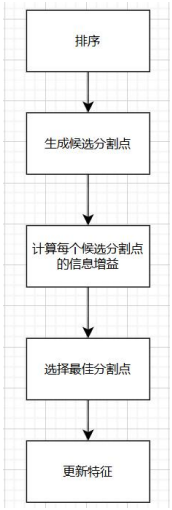
2.7 二分法处理连续值

在决策树算法中，处理连续特征通常采用二分法，将其转化为离散特征。此方法通过寻找最佳分割点，将连续特征划分为两个区间。

1. 原理

二分法的核心思想是将连续值特征转换为离散值，以便于决策树的构建。通过选择合适的分割点，将数据集划分为两个部分，从而减少不纯度（如信息增益或基尼指数）。

2. 实现步骤



（1）排序：将连续特征的所有样本值进行排序，获取唯一值的列表。

（2）生成候选分割点：在排序后的连续值中，每两个相邻值之间生成候选分割点。假设排序后的连续特征值为 x1,x2,…,xn，则候选分割点为

$$\frac{x_i+x_{i+1}}{2} \quad (i = 1, 2, \ldots, n-1)$$

（3）计算每个候选分割点的不纯度：对每个候选分割点，计算该点划分后的加权不纯度。常用的不纯度指标包括信息增益和基尼指数。以信息增益为例，设候选分割点为 p，划分后得到的子集为 D1 和 D2 ，则信息增益计算为：

$$IG(D,p) = H(D) - \left(\frac{|D_1|}{|D|}H(D_1) + \frac{|D_2|}{|D|}H(D_2)\right)$$

其中，H(D)是数据集 D 的熵，｜D1｜和｜D2｜分别是划分后两个子集的样本数量。

（4）选择最佳分割点：比较所有候选分割点的信息增益，选择信息增益最大的分割点作为最终分割点。

（5）更新特征：将连续特征按照最佳分割点划分为两个类别（如：x≤p 和 x>p），并使用离散化后的特征继续构建决策树。

2.8 不使用递归的决策树生成算法

利用队列 queue ，实现层次遍历（广度优先遍历），逐步处理每个节点来建立子树结构。再构建一个辅助队列，将每个节点存储到 nodes_to_process 列表中，以便在树生成完成后可以反向遍历计算每个节点的 leaf_num（叶子节点数量）。对于每个节点，根据特征

选择和树的条件构建子节点；如果达到叶节点条件，直接将其标记为叶节点。最后，逆序处理计算每个结点的叶节点数量：通过逆序遍历 nodes_to_process 列表（即从叶节点到根节点），每次更新父节点的 leaf_num 为其所有子节点 leaf_num 的总和。

输入：训练集 D={$(x_1,y_1),(x_2,y_2),\cdots,(x_m,y_m)$};
属性集合 A={$a_1,a_2,\cdots,a_d$}。
过程：函数 TreeGenerateNonRecursive(D, A)

```
1: 初始化根节点 root 并将 (root, D, A) 入队 queue。
2: 初始化列表 nodes_to_process 用于记录节点的处理顺序。
3: while queue 不为空 do
4:     从 queue 中出队 (current_node, D, A)。
5:     将 current_node 添加到 nodes_to_process。
6:     if D 中所有样本属于同一类 C then
7:         将 current_node 标记为 C 类的叶结点； continue
8:     if A 为空或 D 中样本在 A 上取值相同 then
9:         将 current_node 标记为叶节点，其类别标记为 D 中样本数最多的类； continue
10:    从 A 中选择最佳划分属性a*。
11:    for a*的每个取值a*^v do
12:        生成一个子节点 child_node，对应a*取值为a*^v的样本子集D_v。
13:        if D_v为空 then
14:            将 child_node 标记为叶节点，类别标记为 D 中样本数最多的类； continue
15:        else
16:            将 (child_node,D_v,A\{a*}) 入队 queue。
17:        end if
18:    end for
19: end while
20: 逆序遍历 nodes_to_process，计算每个节点的 leaf_num。
21: return root
```

在构建决策树的过程中，每个节点都会根据特征选择和树的构建条件来决定是否进一步分裂。以下是这个步骤的详细说明：

1、当前节点的特征选择

对于每个节点 current_node，需要从剩余的特征集合 A 中选择一个“最优特征” a* ，用于将数据集 D 划分成不同的子集。这个“最优特征”由基尼指数、信息增益或信息增益率等来确定，使得划分后的子集在类别上更加纯净。

2、判断是否满足叶节点条件

在进一步构建子节点之前，检查当前节点是否满足叶节点条件。如果满足以下任一条件，则将 current_node 标记为叶节点，而不再继续分裂：

（1）**单一类别**：如果数据集 D 中的所有样本都属于同一类 C，则不再需要进一步划分。此时可以将 current_node 标记为叶节点，类别为 C。

（2）**属性集为空或样本在剩余特征上取值相同**：如果 A 为空（即没有剩余特征可以选择），或数据集 D 中样本在剩余特征上的取值都相同，那么即使进一步分裂也不能提供更多信息。在这种情况下，current_node 也被标记为叶节点，并根据 D 中的样本数最多的类别作为 current_node 的类别。

（3）**达到最大深度**：如果当前节点的深度已经达到了预设的最大深度 MaxDepth，则停止继续分裂，将 current_node 直接标记为叶节点，并将类别设为当前数据集中样本数最多的类别。

3、构建子节点

如果不满足叶节点条件，则 current_node 将根据选择的特征 a* 来生成子节点。分情况处理：

（1）当前特征为离散值：如果 a* 是一个离散特征，节点会针对 a* 的每个可能的取值创建一个子节点 child_node，表示 a* 取该值的样本子集。将数据集中所有在 a* 上取值为 a_*^v 的样本（记作$\mathrm{D}_{a_*=a_*^v}$）分配到 child_node，并继续构建树。

如果$\mathrm{D}_{a_*=a_*^v}$为空，即该子集没有样本，说明该特征值在当前分支下没有样本。此时，将 child_node 标记为叶节点，并将其类别设为当前数据集中出现次数最多的类别。

如果$\mathrm{D}_{a_*=a_*^v}$不为空，则将 child_node 和该子集继续加入到构建队列中。

（2）当前特征为连续值：如果 a* 是一个连续特征，则会根据分割点（采用二分法选取）将数据集划分为两个子集。构建两个子节点：一个子节点代表 a* ≥split_valuea 的样本子集；另一个子节点代表 a* <split_valuea 的样本子集。将两个子节点及其对应的数据集加入到构建队列中，继续后续的树构建。

4、将子节点添加到树中

每个 child_node 会作为 current_node 的子节点，存储在 current_node.subtree 中。通过这种方式，不断将子节点加入树中，直到所有节点都满足叶节点条件，不再继续分裂为止。

5、完成子节点分裂后的后续处理

当队列中所有节点都处理完后，逆序遍历已处理的节点列表，计算每个节点的叶节点数。

三、算法设计思路

3.1 三个题目整体的实现思路

首先将决策树算法的框架搭建好，编写节点类、决策树类。决策树类中可以选择不同的划分方式（信息增益、信息增益率等）、剪枝方式（不剪枝、预剪枝、后剪枝），同时可以根据变量是连续变量还是离散变量采取不同的处理方式（如果当前特征或属性为连续变量，则采用二分法进行离散化）。再分别编写函数实现信息增益算法以及信息增益率算法、基尼系数算法，在建树过程中可以递归调用。另外实现画图代码，其中定义多个功能函数，将训练生成的决策树可视化。由于两道选做题包含了剪枝模块，故另外实现了剪枝模块，分别是预剪枝与后剪枝。

在每一次训练时，可以设置不同的训练条件，为决策树类对象设置不同的划分方式、剪枝方式等，同时可以设置训练集测试集的比例，在不同的数据集上训练，最终调用画图模块实现决策树的可视化。

3.2 决策树类的实现思路

需要实现一个生成决策树的函数，采用递归的思想，利用字典的形式记录整个决策树，然后通过不断的递归调用，在字典中添加新的字典，也就是子树。

对于划分方式的实现，首先实现一个计算信息增益的函数，先计算数据集的熵，然后对每个特征进行划分，计算每个子集的熵，再根据熵的变化计算信息增益。在信息增益的基础

上，添加对划分信息的计算，以避免偏向于特征数量较多的情况。然后实现一个计算基尼指数的函数，首先计算每个类的概率，然后根据概率计算基尼值。最后，在决策树的类中设计一个统一的接口，使得在训练时可以选择不同的划分方式。

3.3 不使用递归的建树算法的实现思路

创建两个队列，分别为 queue 与 nodes_to_process。queue = deque([(root, X, y)]) 用来存储节点和数据，queue 的结构为三元组，分别为根节点、当前节点的 X 值，即去除a_*属性后剩下的 X 值，以及 y 标签。nodes_to_process = [] 记录所有节点以便后续计算 leaf_num。

遍历 queue 队列，创建根节点并将其放入队列，并将当前节点存入 nodes_to_process 以记录节点。使用 queue 按层次处理每个节点。

每次处理时，首先检查是否达到叶节点条件（如最大深度或单一类别），如果是则标记为叶节点。如果不是叶节点，则选择最佳分割特征，并根据特征类型（离散或连续）生成对应的子节点。

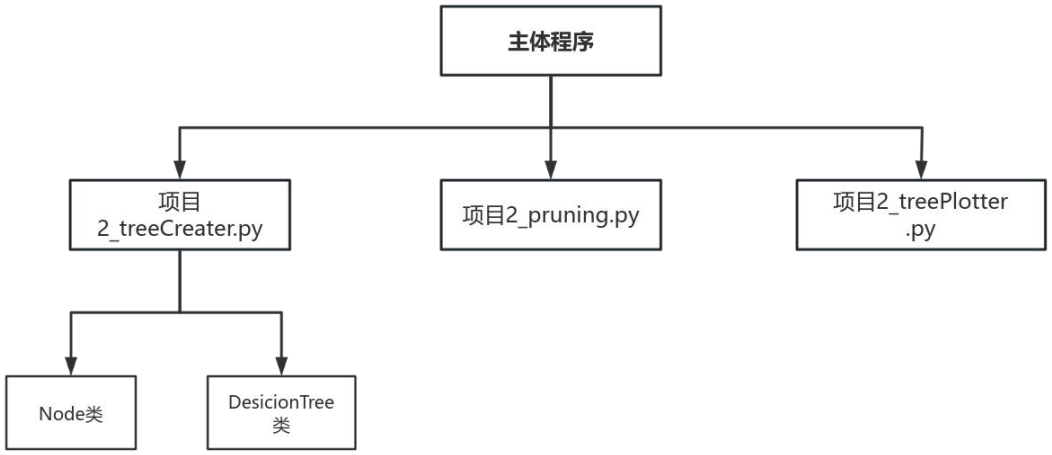
queue 队列处理完毕后，通过 nodes_to_process 逆序遍历，每个节点的 leaf_num 设为其子节点的 leaf_num 总和。

3.4 剪枝部分的实现思路

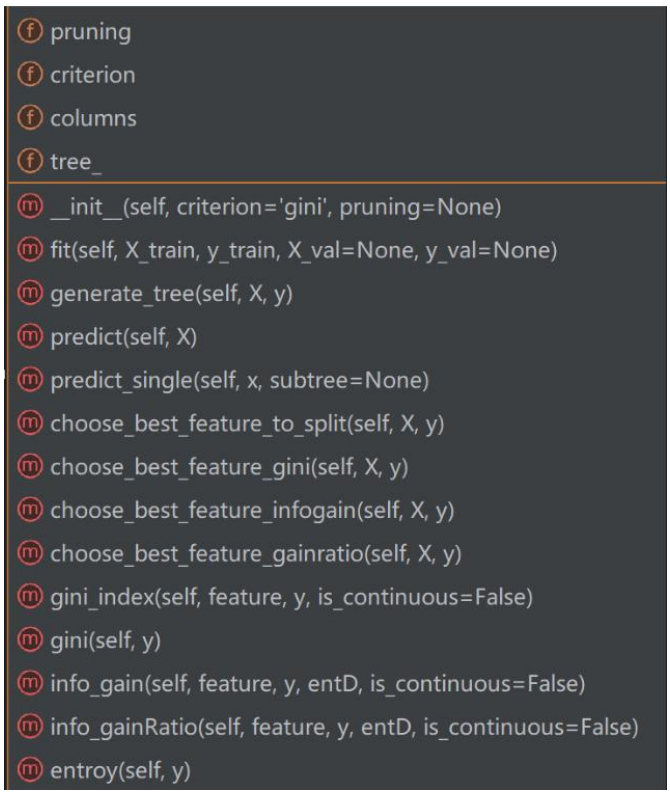
预剪枝：直接从根节点也就是最外层的字典开始递归，然后在每一层的函数中对比剪枝和没有剪枝的准确率，如果替换后的树在验证集上的性能提高或不降低，则执行剪枝操作。预剪枝是从根节点开始，而后剪枝是从最底层的属性节点开始计算：从树的叶子节点开始，逐步向上遍历每个节点。

四、代码结构及核心部分介绍

4.1 整体代码结构



1.DesicionTree 类的成员以及方法函数



下面逐一介绍各自的方法及作用：

成员变量：

criterion	用于选择特征划分的方法。可选值为 'gini'（基尼系数）、'infogain'（信息增益）和 'gainratio'（信息增益率）。
pruning	剪枝方法的选择。可选值为 None（不剪枝）、'pre_pruning'（预剪枝）和 'post_pruning'（后剪枝）。
columns	数据集中所有特征的列名列表。
tree_	生成的决策树，是一个 Node 对象。

方法函数：

（1）__init__(self, criterion='gini', pruning=None):

参数：

criterion: 特征划分方法，默认为 'gini'。

pruning: 剪枝方法，默认为 None。

作用：为 DesicionTree 类的构造函数，设置划分方法和剪枝方法。

（2）fit(self, X_train, y_train, X_val=None, y_val=None):

参数:

X_train: 训练集特征数据，pd.DataFrame 类型。

y_train: 训练集标签数据，pd.Series 类型。

X_val: 验证集特征数据，pd.DataFrame 类型，用于剪枝。

y_val: 验证集标签数据，pd.Series 类型，用于剪枝。

作用: 根据传入的训练集以及测试集（如果有剪枝操作的话则传入验证集）训练决策树模型，生成决策树。如果启用了剪枝，则根据验证集进行剪枝。

（3）generate_tree(self, X, y):

参数:

X: 当前节点的特征数据，pd.DataFrame 类型。

y: 当前节点的标签数据，pd.Series 类型。

作用: generate_tree 函数是决策树生成的核心函数，用于递归地构建决策树。它根据输入的特征数据 X 和标签数据 y，选择最佳特征进行划分，并生成子树。最终返回一个完整的决策树。

（4）predict(self, X):

参数:

X: 待预测的特征数据，pd.DataFrame 类型。

作用: predict 函数用于对输入的特征数据 X 进行预测，返回预测结果。该函数支持批量预测和单样本预测。

实现思路

1、检查模型是否已训练：在预测之前，检查是否已经生成了决策树模型 tree_。如果没有，则抛出异常。

2、单样本预测：如果输入数据 X 是一个单样本（即 X.ndim == 1），则调用 predict_single 函数进行预测。

3、批量预测：如果输入数据 X 是一个数据框（即 X.ndim > 1），则对每一行数据调用 predict_single 函数进行预测，并返回预测结果。

（5）predict_single(self, x, subtree=None)：

参数

x: 单个样本的特征数据，pd.Series 类型。

subtree: 当前子树，Node 类型。默认值为 None，表示从根节点开始遍历。

作用：predict_single 函数用于对单个样本 x 进行预测，递归遍历决策树，返回预测结果。该函数是决策树预测的核心函数之一。

实现思路

1、初始化子树：如果 subtree 为 None，则从根节点开始遍历。

2、检查是否为叶子节点：如果当前节点是叶子节点（即 subtree.is_leaf 为 True），则返回叶子节点的类别 subtree.leaf_class。

3、递归遍历子树

如果当前特征是连续值（即 subtree.is_continuous 为 True），则根据特征值 x[subtree.feature_index] 与分割点 subtree.split_value 的关系，选择相应的子树进行递归预测。

如果当前特征是离散值（即 subtree.is_continuous 为 False），则根据特征值 x[subtree.feature_index] 选择相应的子树进行递归预测。

（6）choose_best_feature_to_split(self, X, y)

参数：

X: 当前节点的特征数据，pd.DataFrame 类型。

y: 当前节点的标签数据，pd.Series 类型。

作用：根据 self.criterion 的值，调用相应的划分方法函数。如果 self.criterion 为 'gini'，则调用 choose_best_feature_gini 函数。如果 self.criterion 为 'infogain'，则调用 choose_best_feature_infogain 函数。如果 self.criterion 为 'gainratio'，则调用 choose_best_feature_gainratio 函数。

（7）choose_best_feature_gini(self, X, y):

参数:

X: 当前节点的特征数据，pd.DataFrame 类型。

y: 当前节点的标签数据，pd.Series 类型。

作用: choose_best_feature_gini 函数用于根据基尼系数选择最佳特征进行划分，通过遍历地计算每个特征的基尼系数，选择基尼系数最小的特征作为最佳划分特征。

函数实现

1、初始化最佳特征和基尼系数：初始化 best_feature_name 和 best_gini，用于记录最佳特征的名称和对应的基尼系数。

2、遍历所有特征：遍历 X 中的每个特征，计算其基尼系数。

3、判断特征是否为连续值：使用 type_of_target 函数判断当前特征是否为连续值。

4、计算基尼系数：调用 gini_index 函数计算当前特征的基尼系数。

5、更新最佳特征：如果当前特征的基尼系数小于 best_gini，则更新 best_feature_name 和 best_gini。

6、返回最佳特征和基尼系数：返回最佳特征的名称和对应的基尼系数。

（8）choose_best_feature_infogain(self, X, y):

参数:

X: 当前节点的特征数据，pd.DataFrame 类型。

y: 当前节点的标签数据，pd.Series 类型。

作用: choose_best_feature_infogain 函数用于根据信息增益选择最佳特征进行划分。函数通过计算每个特征的信息增益，选择信息增益最大的特征作为最佳划分特征。

函数实现

1、计算数据集的信息熵：调用 entroy 函数计算当前数据集 y 的信息熵 entD。

2、初始化最佳特征和信息增益：初始化 best_feature_name 和 best_info_gain，用于记录最佳特征的名称和对应的信息增益。

3、遍历所有特征：遍历 X 中的每个特征，计算其信息增益。

4、判断特征是否为连续值：使用 type_of_target 函数判断当前特征是否为连续值。

5、信息增益：调用 info_gain 函数计算当前特征的信息增益。

6、更新最佳特征：如果当前特征的信息增益大于 best_info_gain，则更新 best_feature_name 和 best_info_gain。

7、返回最佳特征和信息增益：返回最佳特征的名称和对应的信息增益。

（9）choose_best_feature_gainratio(self, X, y):

参数:

X: 当前节点的特征数据，pd.DataFrame 类型。

y: 当前节点的标签数据，pd.Series 类型。

作用: choose_best_feature_gainratio 函数用于根据信息增益率选择最佳特征进行划分。函数通过计算每个特征的信息增益率，选择信息增益率最大的特征作为最佳划分特征。实现过程与 choose_best_feature_infogain 函数类似。

（10）gini_index(self, feature, y, is_continuous=False):

参数:

feature: 当前特征的值，pd.Series 类型。

y: 当前节点的标签数据，pd.Series 类型。
is_continuous: 当前特征是否为连续值，bool 类型。
作用: gini_index 函数用于计算基尼指数，用于选择最佳特征进行划分。函数根据特征是否为连续值，分别计算离散特征和连续特征的基尼指数。若为连续特征，同时还返回一个分割点。

（11）gini(self, y):

参数:
y: 当前节点的标签数据，pd.Series 类型。
描述: 根据基尼系数的计算公式计算基尼系数。

```
def gini(self, y):
    p = pd.value_counts(y) / y.shape[0]
    gini = 1 - np.sum(p ** 2)
    return gini
```

（12）info_gain(self, feature, y, entD, is_continuous=False):

参数:
feature: 当前特征的值，pd.Series 类型。
y: 当前节点的标签数据，pd.Series 类型。
entD: 当前节点的信息熵。
is_continuous: 当前特征是否为连续值，bool 类型。
作用: info_gain 函数用于计算信息增益，用于选择最佳特征进行划分。函数根据特征是否为连续值，分别计算离散特征和连续特征的信息增益。

（13）info_gainRatio(self, feature, y, entD, is_continuous=False):

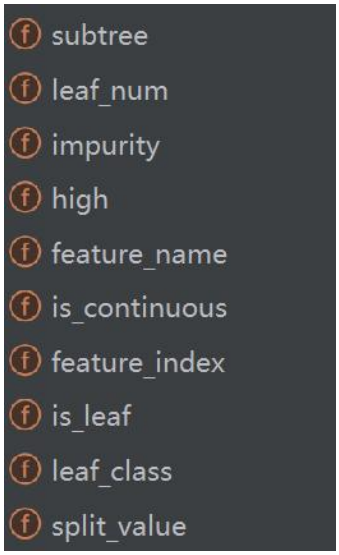
参数:
feature: 当前特征的值，pd.Series 类型。
y: 当前节点的标签数据，pd.Series 类型。
entD: 当前节点的信息熵。
is_continuous: 当前特征是否为连续值，bool 类型。
作用: info_gainRatio 函数用于计算信息增益率，用于选择最佳特征进行划分。函数根据特征是否为连续值，分别计算离散特征和连续特征的信息增益率。其实现与 info_gain 函数类似，故“核心部分”只展示 info_gain 函数。

（14）entroy(self, y):

参数:
y: 当前节点的标签数据，pd.Series 类型。
作用: 根据信息熵公式计算信息熵，并返回。

```
def entroy(self, y):    #计算信息熵
    p = pd.value_counts(y) / y.shape[0]  # 计算各类样本所占比率
    ent = np.sum(-p * np.log2(p))
    return ent
```

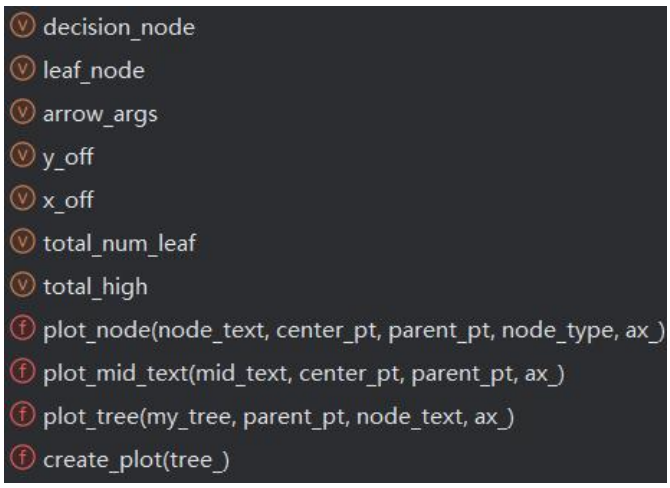
2.Node 类的成员变量：



成员变量：

- feature_name: 当前节点选用的划分属性（依据）的特征名称。
- feature_index: 当前节点选用的划分属性的特征索引。
- subtree: 字典类型，当前节点的子树，键为特征值或分割点，值为子节点。
- impurity: 当前节点的度量值（基尼系数、信息增益等）。
- is_continuous:布尔类型，表示当前节点的属性是否为连续值。
- split_value: 当前节点特征为连续值时的分割点（本程序采用二分法处理连续值）。
- is_leaf: 当前节点是否为叶子节点。
- leaf_class: 当前叶子节点的类别，也就是标签值。
- leaf_num: 当前节点及其子树中叶子节点的总数。
- high: 当前节点的高度。

（2）Tree_Ploter.py 的变量及函数



全局变量：

y_off: 用于控制节点在垂直方向上的偏移量。
x_off: 用于控制节点在水平方向上的偏移量。
total_num_leaf: 决策树中叶子节点的总数。
total_high: 决策树的总高度。

方法函数：

（1）plot_node(node_text, center_pt, parent_pt, node_type, ax_): 绘制节点和箭头。

（2）plot_mid_text(mid_text, center_pt, parent_pt, ax_): 在节点之间绘制中间文本。

（3）plot_tree(my_tree, parent_pt, node_text, ax_):

参数：

my_tree: 当前节点。

parent_pt: 父节点坐标。

node_text: 节点文本。

ax_: 绘图对象。

功能：计算当前节点的中心坐标 center_pt。

绘图流程：

1、绘制中间文本 node_text。

2、如果 total_high 为 0，表示当前节点是叶子节点，直接绘制叶子节点。

3、否则，绘制决策节点，并递归绘制子节点。

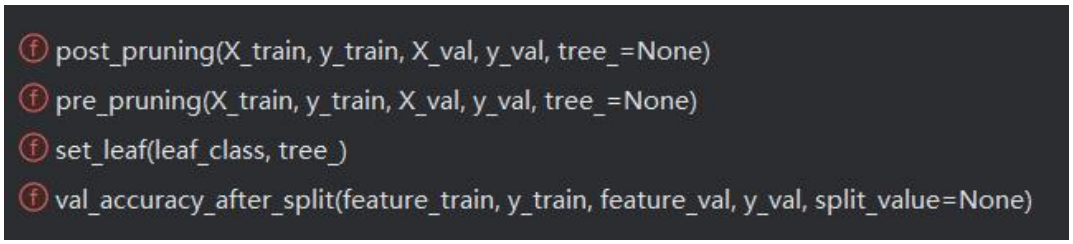
4、更新 y_off 和 x_off，以便在递归过程中正确放置节点。

```
def plot_tree(my_tree, parent_pt, node_text, ax_):
    global y_off
    global x_off
    global total_num_leaf
    global total_high

    num_of_leaf = my_tree.leaf_num
    center_pt = (x_off + (1 + num_of_leaf) / (2 * total_num_leaf), y_off)

    plot_mid_text(node_text, center_pt, parent_pt, ax_)

    if total_high == 0:  # total_high 为零时，表示就直接为一个叶节点。因为西瓜数据集的原因，在预剪枝的时候，有时候会遇到这种情况。
        plot_node(my_tree.leaf_class, center_pt, parent_pt, leaf_node, ax_)
        return
    plot_node(my_tree.feature_name, center_pt, parent_pt, decision_node, ax_)

    y_off -= 1 / total_high
    for key in my_tree.subtree.keys():
        if my_tree.subtree[key].is_leaf:
            x_off += 1 / total_num_leaf
            plot_node(str(my_tree.subtree[key].leaf_class), (x_off, y_off), center_pt, leaf_node, ax_)
            plot_mid_text(str(key), (x_off, y_off), center_pt, ax_)
        else:
            plot_tree(my_tree.subtree[key], center_pt, str(key), ax_)
    y_off += 1 / total_high
```

（4）create_plot(tree_): 创建并显示决策树的可视化图。

```
def create_plot(tree_):
    global y_off
    global x_off
    global total_num_leaf
    global total_high

    total_num_leaf = tree_.leaf_num
    total_high = tree_.high
    y_off = 1
    x_off = -0.5 / total_num_leaf

    fig_, ax_ = plt.subplots(figsize=(10, 10))  # 设置画布大小
    ax_.set_xticks([])  # 隐藏坐标轴刻度
    ax_.set_yticks([])
    ax_.spines['right'].set_color('none')  # 设置隐藏坐标轴
    ax_.spines['top'].set_color('none')
    ax_.spines['bottom'].set_color('none')
    ax_.spines['left'].set_color('none')
    plot_tree(tree_, (0.5, 1), '', ax_)

    plt.show()
```

3. pruning.py 的变量及函数



函数功能介绍：

（1）post_pruning(X_train, y_train, X_val, y_val, tree_=None): 后剪枝函数，参数分别为：特征值训练集、标签值训练集、特征值验证集以及标签值验证集。验证集用于计算剪枝时的当前节点准确率，详细介绍见“核心部分介绍模块”。

（2）pre_pruning(X_train, y_train, X_val, y_val, tree_=None): 预剪枝函数，参数分别为：特征值训练集、标签值训练集、特征值验证集以及标签值验证集。验证集用于计算剪枝时的当前节点准确率，详细介绍见“核心部分介绍模块”。

（3）set_leaf(leaf_class, tree_):

参数：

leaf_class: 叶子节点的类别。

tree_: 当前节点，Node 类型。

功能：将当前节点设置为叶子节点，并更新相关属性。

（4）val_accuracy_after_split(feature_train, y_train, feature_val, y_val, split_value=None):

参数：

feature_train: 训练集特征数据，pd.Series 类型。

y_train: 训练集标签数据，pd.Series 类型。

feature_val: 验证集特征数据，pd.Series 类型。

y_val: 验证集标签数据，pd.Series 类型。

split_value: 分割点，float 类型，默认为 None。

功能：

如果 split_value 不为 None，则按分割点对特征进行分组。

计算训练集中各特征下样本最多的类别 majority_class_in_train。

计算验证集中各类别对应的数量 right_class_in_val。

返回验证集的准确率。

4.2 核心部分介绍

4.2.1 generate_tree(self, X, y):

（1）实现思路

首先初始化节点，创建一个新的 Node 对象 my_tree，并初始化叶子节点数量 leaf_num 为 0。

然后检查终止条件，如果所有样本属于同一类别（当前节点的所有样本都属于同一类别），则将该节点标记为叶子节点，并设置其类别。或者特征用完，当前节点的特征数据为空（即所有特征都已用于划分），则将该节点标记为叶子节点，并设置其类别为样本数最多的类别。

接着选择最佳特征。调用 choose_best_feature_to_split 函数，选择最佳特征进行划分，并获取最佳特征的名称 best_feature_name 和杂质度量 best_impurity。

并设置节点属性，将当前节点的特征名称 feature_name 设置为 best_feature_name，杂质度量 impurity 设置为 best_impurity[0]，特征索引 feature_index 设置为 best_feature_name 在 self.columns 中的索引。

找到最佳特征后，划分数据集。如果该特征为离散值特征，即 best_impurity 的长度为 1，表示当前特征为离散值。遍历特征的所有唯一值，递归生成子树，并记录子树的高度和叶子节点数量。若为连续值特征，即 best_impurity 的长度为 2，表示当前特征为连续值。则利用二分法，根据最佳分割点将数据集划分为两部分，递归生成子树，并记录子树的高度和叶子节点数量。

最后返回生成的节点 my_tree。

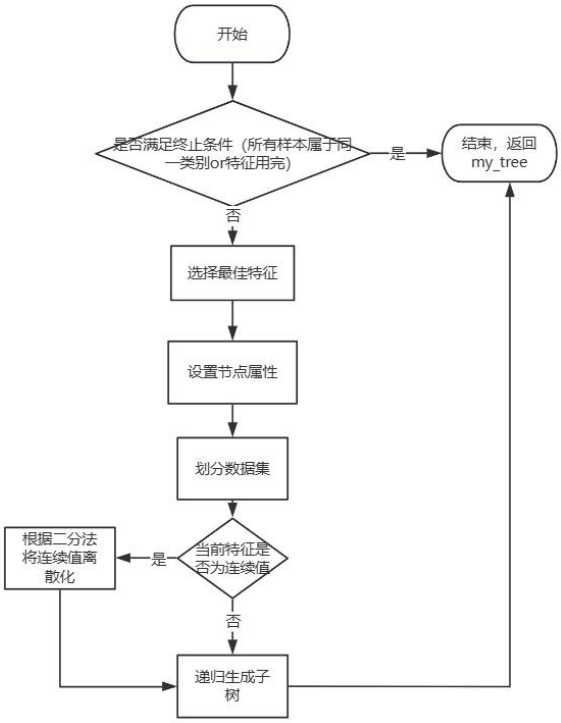


图 2 generate_tree 函数的实现思路

（2）实现方式

递归生成子树：通过递归调用 generate_tree 函数，生成子树。每次递归时，根据当前节点的特征值划分数据集，并生成新的子节点。

终止条件检查：在每次递归时，检查是否满足终止条件（所有样本属于同一类别或特征用完），如果满足则停止递归，返回叶子节点。

记录节点信息：在生成子树的过程中，记录每个节点的特征名称、杂质度量、特征索引、是否为连续值、分割点、是否为叶子节点、叶子节点类别、叶子节点数量和高度等信息。

4.2.2 gini_index(self, feature, y, is_continuous=False):

（1）实现思路

首先计算样本总数，获取当前节点的样本总数 m。

然后获取特征的唯一值，也就是去除重复项，将剩下的特征值存入列表 unique_value 中。

接着需要判断特征是否为连续值:

如果为离散值特征，即 is_continuous 为 False，则计算每个特征值对应的基尼系数，并累加得到基尼指数。直接返回总的基尼系数

如果为连续值特征，即 is_continuous 为 True，则对该连续特征进行二分法离散化：对特征值进行排序，并生成所有可能的分割点。遍历所有分割点，计算每个分割点的基尼指数，选择基尼指数最小的分割点。最后返回一个列表，第一个值为基尼系数，第二个值为连续变量的分割点。

（2）离散值特征与连续值特征的源代码

离散值特征：

```
if not is_continuous:
    gini_index = 0
```

```
    for value in unique_value:
        Dv = y[feature == value]
        m_dv = Dv.shape[0]  #筛选出特征值等于 value 的样本 Dv，并计算其样本数 m_dv
        gini = self.gini(Dv)  #调用 gini 函数计算 Dv 的基尼系数 gini
        gini_index += m_dv / m * gini
    return [gini_index]
```

连续值特征：

```
else:
    unique_value.sort() #对 unique_value 进行排序。
    split_point_set = [(unique_value[i] + unique_value[i + 1]) / 2 for i in range(len(unique_value) - 1)]  #生成所有可能的分割点 split_point_set，即每两个相邻特征值的中点
    min_gini = float('inf')
    min_gini_point = None
    for split_point in split_point_set:  #筛选出特征值小于等于 split_point 的样本 Dv1 和大于 split_point 的样本 Dv2
        Dv1 = y[feature <= split_point]
        Dv2 = y[feature > split_point]
        gini_index = Dv1.shape[0] / m * self.gini(Dv1) + Dv2.shape[0] / m * self.gini(Dv2)
        if gini_index < min_gini:
            min_gini = gini_index
            min_gini_point = split_point
    return [min_gini, min_gini_point]  #返回基尼指数和最佳分割点
```

4.2.3 info_gain 函数的实现

（1）实现思路

首先，计算样本总数：获取当前节点的样本总数 m。

然后获取特征的唯一值，也就是去除重复项，将剩下的特征值存入列表 unique_value 中。

接着需要判断特征是否为连续值:

如果为离散值特征，即 is_continuous 为 False，则计算每个特征值对应的信息熵，并累加得到特征的信息熵 feature_ent。然后计算信息增益 gain，并直接返回。

如果为连续值特征，即 is_continuous 为 True，则对特征值进行排序，并生成所有可能的分割点。遍历所有分割点，计算每个分割点的信息熵，选择信息熵最小的分割点。然后计算信息增益 gain。并返回信息增益值 gain 以及连续值划分点（二分法划分）。

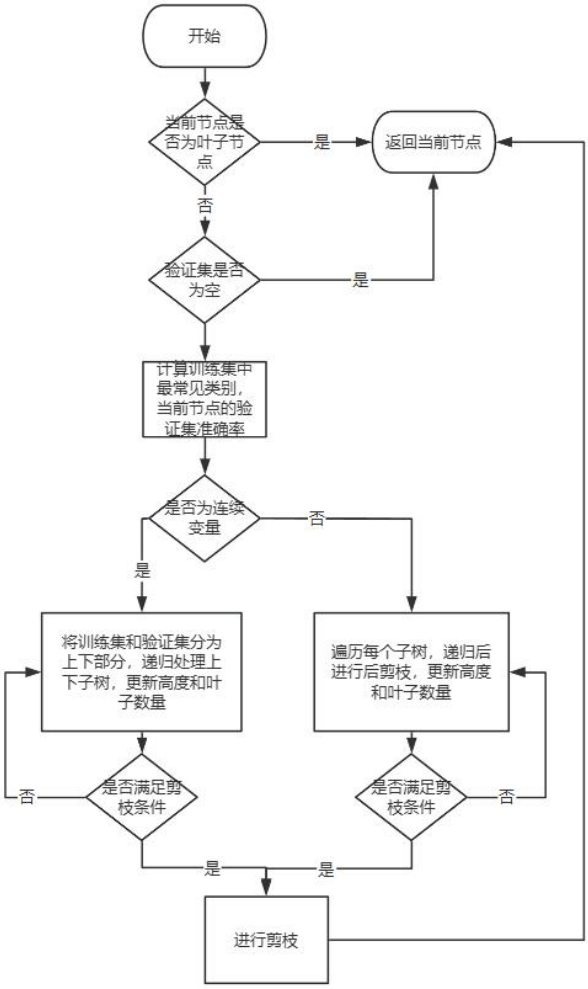
（2）源代码及详细注释

```
def info_gain(self, feature, y, entD, is_continuous=False):
    '''
    :param feature: 当前特征（属性）下所有样本值
    :param y:       对应标签值
    :return:        当前特征的信息增益，list 类型，若当前特征为离散值则只有一个元素为信息增益，若为连续值，则第一个元素为信息增益，第二个元素为切分点
    '''
    m = y.shape[0]      #y 的 shape[0]是行数，也就是样本数
```

```
    unique_value = pd.unique(feature)       #属性 a 的 av，例如：颜色有乌黑、青绿等
    if is_continuous:  #如果是连续值的话，需要进行二分（离散化），要根据最小信息熵找出最佳划分点
        unique_value.sort()  # 排序, 用于建立分割点
        split_point_set = [(unique_value[i] + unique_value[i + 1]) / 2 for i in range(len(unique_value) - 1)]  #在每两个值的区间取中点
        min_ent = float('inf')  # 挑选信息熵最小的分割点
        min_ent_point = None
        for split_point_ in split_point_set:

            Dv1 = y[feature <= split_point_]  #这是第一类的标签值
            Dv2 = y[feature > split_point_]   #这是第二类的标签值
            feature_ent_ = Dv1.shape[0] / m * self.entroy(Dv1) + Dv2.shape[0] / m * self.entroy(Dv2)          #套公式，信息增益公式负号后面的部分，越小越好

            if feature_ent_ < min_ent:
                min_ent = feature_ent_
                min_ent_point = split_point_
        gain = entD - min_ent    #信息增益公式

        return [gain, min_ent_point]  #返回该特征（属性）的信息增益值以及连续值划分点（二分）

    else:     #该特征为离散值
        feature_ent = 0       #信息增益公式负号后面的部分，越小越好
        for value in unique_value:    #遍历 av
            Dv = y[feature == value]  # 当前特征中取值为 value 的样本，即书中的 D^{v}
            feature_ent += Dv.shape[0] / m * self.entroy(Dv)

        gain = entD - feature_ent  # 信息增益公式
        return [gain]
```

4.2.4 后剪枝函数 post_pruning

函数实现思路及流程如下图，具体可见详细注释




```
def post_pruning(X_train, y_train, X_val, y_val, tree_=None):
    """
    后剪枝函数，用于对决策树进行后剪枝操作。

    参数:
    X_train: 训练集特征数据，pd.DataFrame 类型。
    y_train: 训练集标签数据，pd.Series 类型。
    X_val: 验证集特征数据，pd.DataFrame 类型。
    y_val: 验证集标签数据，pd.Series 类型。
    tree_: 当前节点，Node 类型。

    返回:
    剪枝后的决策树节点。
    """
    # 如果当前节点已经是叶子节点，直接返回
    if tree_.is_leaf:
        return tree_

    # 如果验证集为空，不再剪枝，直接返回
    if X_val.empty:
```

```
        return tree_

    # 计算训练集中样本最多的类别
    most_common_in_train = pd.value_counts(y_train).index[0]
    # 计算当前节点下验证集样本的准确率
    current_accuracy = np.mean(y_val == most_common_in_train)

    # 如果当前节点是连续值特征
    if tree_.is_continuous:
        # 根据分割点将训练集和验证集分为两部分
        up_part_train = X_train.loc[:, tree_.feature_name] >= tree_.split_value
        down_part_train = X_train.loc[:, tree_.feature_name] < tree_.split_value
        up_part_val = X_val.loc[:, tree_.feature_name] >= tree_.split_value
        down_part_val = X_val.loc[:, tree_.feature_name] < tree_.split_value

        # 递归处理上部分子树
        up_subtree = post_pruning(X_train[up_part_train], y_train[up_part_train], X_val[up
_part_val],
                                  y_val[up_part_val],
                                  tree_.subtree['>= {:.3f}'.format(tree_.split_value)])
        tree_.subtree['>= {:.3f}'.format(tree_.split_value)] = up_subtree

        # 递归处理下部分子树
        down_subtree = post_pruning(X_train[down_part_train], y_train[down_part_train],
                                    X_val[down_part_val], y_val[down_part_val],
                                    tree_.subtree['< {:.3f}'.format(tree_.split_value)])
        tree_.subtree['< {:.3f}'.format(tree_.split_value)] = down_subtree

        # 更新当前节点的高度和叶子节点数量
        tree_.high = max(up_subtree.high, down_subtree.high) + 1
        tree_.leaf_num = (up_subtree.leaf_num + down_subtree.leaf_num)

        # 如果上部分和下部分子树都是叶子节点
        if up_subtree.is_leaf and down_subtree.is_leaf:
            # 定义分割函数
            def split_fun(x):
                if x >= tree_.split_value:
                    return '>= {:.3f}'.format(tree_.split_value)
                else:
                    return '< {:.3f}'.format(tree_.split_value)

            # 根据分割函数对验证集进行分割
            val_split = X_val.loc[:, tree_.feature_name].map(split_fun)
            # 计算分割后的验证集准确率
```

```
            right_class_in_val = y_val.groupby(val_split).apply(
                lambda x: np.sum(x == tree_.subtree[x.name].leaf_class))
            split_accuracy = right_class_in_val.sum() / y_val.shape[0]

            # 如果当前节点为叶子节点时的准确率大于不剪枝的准确率，则进行剪枝操作
            if current_accuracy > split_accuracy:
                set_leaf(pd.value_counts(y_train).index[0], tree_)
    else:
        # 初始化最大高度和叶子节点数量
        max_high = -1
        tree_.leaf_num = 0
        is_all_leaf = True  # 判断当前节点下，所有子树是否都为叶节点

        # 遍历当前节点的所有子树
        for key in tree_.subtree.keys():
            # 根据特征值将训练集和验证集分为两部分
            this_part_train = X_train.loc[:, tree_.feature_name] == key
            this_part_val = X_val.loc[:, tree_.feature_name] == key

            # 递归处理子树
            tree_.subtree[key] = post_pruning(X_train[this_part_train], y_train[this_part_tra
in],
                                              X_val[this_part_val], y_val[this_part_va
l], tree_.subtree[key])
            # 更新最大高度和叶子节点数量
            if tree_.subtree[key].high > max_high:
                max_high = tree_.subtree[key].high
            tree_.leaf_num += tree_.subtree[key].leaf_num

            # 判断子树是否为叶子节点
            if not tree_.subtree[key].is_leaf:
                is_all_leaf = False
        # 更新当前节点的高度
        tree_.high = max_high + 1

        # 如果所有子节点都为叶子节点，则考虑是否进行剪枝
        if is_all_leaf:
            # 计算分割后的验证集准确率
            right_class_in_val = y_val.groupby(X_val.loc[:, tree_.feature_name]).apply(
                lambda x: np.sum(x == tree_.subtree[x.name].leaf_class))
            split_accuracy = right_class_in_val.sum() / y_val.shape[0]

            # 如果当前节点为叶子节点时的准确率大于不剪枝的准确率，则进行剪枝操作
            if current_accuracy > split_accuracy:
```

```
                set_leaf(pd.value_counts(y_train).index[0], tree_)

    # 返回剪枝后的节点
    return tree_
```

4.2.5 预剪枝函数 pre_pruning

（1）实现思路

在构建决策树的过程中，动态评估每个节点是否应该进行分裂，基于当前节点的准确率和分裂后的准确率进行比较，决定是否将节点设置为叶子节点。

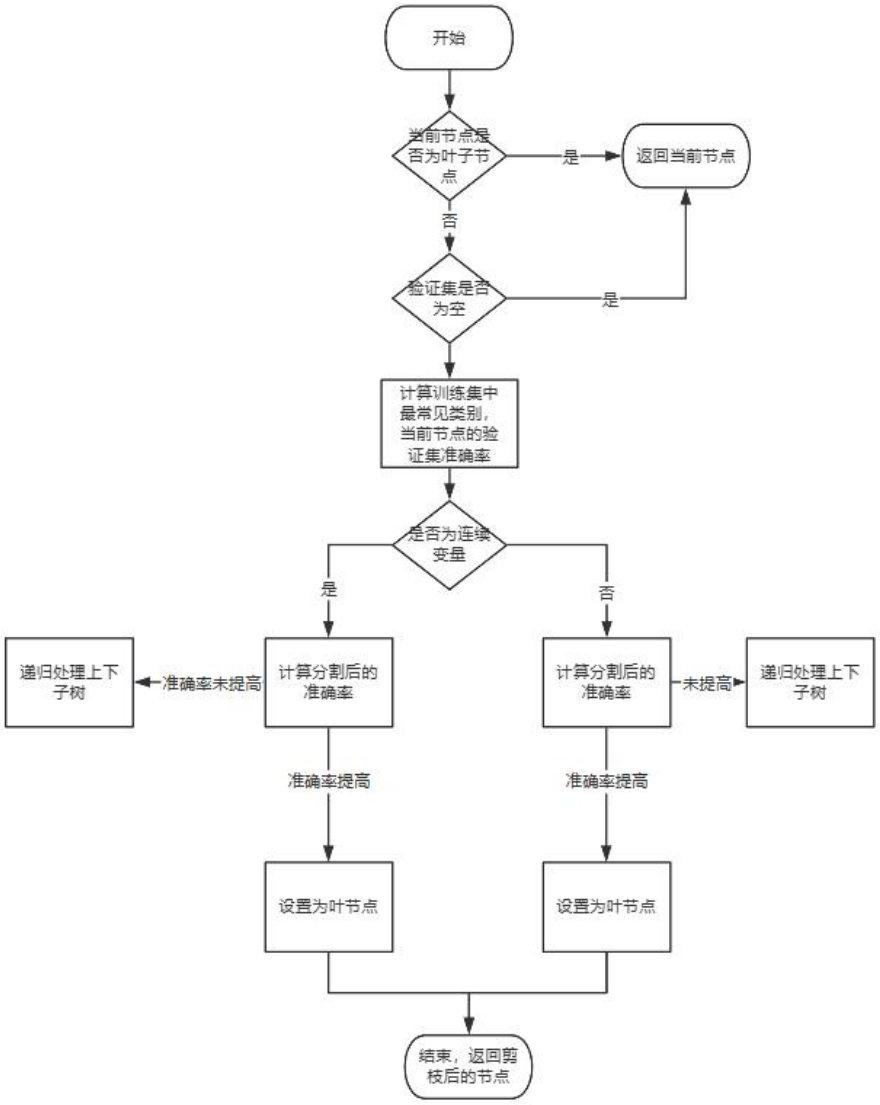
首先检查当前节点是否为叶子节点。如果是，直接返回；如果验证集为空，也不进行剪枝。

然后需要计算准确，计算训练集中最常见类别的准确率，作为当前节点的基准准确率。

特征分为连续特征和离散特征。对于连续特征，计算在当前分割条件下的准确率，判断当前准确率是否高于分割后的准确率。如果当前准确率更高，将当前节点设置为叶子节点；否则，递归处理其左右子树。对于离散特征，类似地计算分割后的准确率，判断是否剪枝，并递归处理每个子树。

最终返回剪枝后的树节点，以此提高模型的泛化能力。

（2）实现流程



（3）详细代码注释

```
def pre_pruning(X_train, y_train, X_val, y_val, tree_=None):
    # 如果当前节点已经是叶节点，直接返回
    if tree_.is_leaf:
        return tree_

    # 如果验证集为空，则不进行剪枝，直接返回当前树
    if X_val.empty:
        return tree_

    # 计算训练集中样本最多的类别
    most_common_in_train = pd.value_counts(y_train).index[0]
    # 计算当前节点在验证集上的准确率
    current_accuracy = np.mean(y_val == most_common_in_train)

    # 如果当前节点是连续特征
```

```
    if tree_.is_continuous:
        # 计算分割后的准确率
        split_accuracy = val_accuracy_after_split(X_train[tree_.feature_name], y_train,
                                                  X_val[tree_.feature_name], y_val,
                                                  split_value=tree_.split_value)

        # 比较当前准确率和分割后的准确率
        if current_accuracy >= split_accuracy:
            # 如果当前准确率更高，则将当前节点设置为叶节点
            set_leaf(pd.value_counts(y_train).index[0], tree_)
        else:
            # 根据分割值将训练集和验证集分为上下两部分
            up_part_train = X_train.loc[:, tree_.feature_name] >= tree_.split_value
            down_part_train = X_train.loc[:, tree_.feature_name] < tree_.split_value
            up_part_val = X_val.loc[:, tree_.feature_name] >= tree_.split_value
            down_part_val = X_val.loc[:, tree_.feature_name] < tree_.split_value

            # 递归处理上部分子树
            up_subtree = pre_pruning(X_train[up_part_train], y_train[up_part_train],
                                     X_val[up_part_val], y_val[up_part_val],
                                     tree_.subtree['>= {:.3f}'.format(tree_.split_value)])
            tree_.subtree['>= {:.3f}'.format(tree_.split_value)] = up_subtree

            # 递归处理下部分子树
            down_subtree = pre_pruning(X_train[down_part_train], y_train[down_part_trai
n],
                                       X_val[down_part_val], y_val[down_part_val],
                                       tree_.subtree['< {:.3f}'.format(tree_.split_valu
e)])
            tree_.subtree['< {:.3f}'.format(tree_.split_value)] = down_subtree

            # 更新当前节点的高度和叶子节点数量
            tree_.high = max(up_subtree.high, down_subtree.high) + 1
            tree_.leaf_num = (up_subtree.leaf_num + down_subtree.leaf_num)

    else:  # 如果是离散特征
        # 计算分割后的准确率
        split_accuracy = val_accuracy_after_split(X_train[tree_.feature_name], y_train,
                                                  X_val[tree_.feature_name], y_val)

        # 比较当前准确率和分割后的准确率
        if current_accuracy >= split_accuracy:
            # 如果当前准确率更高，则将当前节点设置为叶节点
            set_leaf(pd.value_counts(y_train).index[0], tree_)
```

```
        else:
            max_high = -1
            tree_.leaf_num = 0
            # 遍历每个子树
            for key in tree_.subtree.keys():
                # 根据特征值将训练集和验证集分为子部分
                this_part_train = X_train.loc[:, tree_.feature_name] == key
                this_part_val = X_val.loc[:, tree_.feature_name] == key

                # 递归处理子树
                tree_.subtree[key] = pre_pruning(X_train[this_part_train], y_train[this_part
_train],
                                                 X_val[this_part_val], y_val[this_part
_val],
                                                 tree_.subtree[key])

                # 更新最大高度和叶子节点数量
                if tree_.subtree[key].high > max_high:
                    max_high = tree_.subtree[key].high
                tree_.leaf_num += tree_.subtree[key].leaf_num

            # 更新当前节点的高度
            tree_.high = max_high + 1

    # 返回剪枝后的节点
    return tree_
```

4.2.6 非递归建树代码实现

```
def generate_tree(self, X, y):
    root = Node()
    root.high = 0  # 根节点的高度为 0
    queue = deque([(root, X, y)])  # 使用队列来存储节点和数据
    nodes_to_process = []  # 记录所有节点以便后续计算 leaf_num
    while queue:
        current_node, current_X, current_y = queue.popleft()
        nodes_to_process.append(current_node)

        # 叶节点条件：达到最大深度或只有单一类别或没有特征
```

```
        if current_node.high >= self.MaxDepth or current_y.nunique() == 1 or current_X.e
mpty:
            current_node.is_leaf = True
            current_node.leaf_class = current_y.mode()[0]
            current_node.leaf_num = 1  # 是叶子节点，叶子数量为 1
            continue

        # 选择最佳划分特征
        best_feature_name, best_impurity = self.choose_best_feature_to_split(current_X, c
urrent_y)
        current_node.feature_name = best_feature_name
        current_node.impurity = best_impurity[0]
        current_node.feature_index = self.columns.index(best_feature_name)
        feature_values = current_X[best_feature_name]

        if len(best_impurity) == 1:  # 离散值特征
            current_node.is_continuous = False
            unique_vals = feature_values.unique()
            sub_X = current_X.drop(best_feature_name, axis=1)

            for value in unique_vals:
                child_node = Node()
                child_node.high = current_node.high + 1
                queue.append((child_node, sub_X[feature_values == value], current_y[fea
ture_values == value]))
                current_node.subtree[value] = child_node

        elif len(best_impurity) == 2:  # 连续值特征
            current_node.is_continuous = True
```

```
                current_node.split_value = best_impurity[1]
                up_part = '>= {:.3f}'.format(current_node.split_value)
                down_part = '< {:.3f}'.format(current_node.split_value)

                child_node_up = Node()
                child_node_down = Node()
                child_node_up.high = current_node.high + 1
                child_node_down.high = current_node.high + 1
                queue.append((child_node_up, current_X[feature_values >= current_node.split
_value],
                              current_y[feature_values >= current_node.split_value]))
                queue.append((child_node_down, current_X[feature_values < current_node.spli
t_value],
                              current_y[feature_values < current_node.split_value]))

                current_node.subtree[up_part] = child_node_up
                current_node.subtree[down_part] = child_node_down

    # 逆序遍历 nodes_to_process，计算每个节点的 leaf_num
    while nodes_to_process:
        node = nodes_to_process.pop()
        if node.is_leaf:
            node.leaf_num = 1
        else:
            node.leaf_num = sum(child.leaf_num for child in node.subtree.values())

    return root
```

五、题目 1.1 实验流程、测试结果及分析

5.1 流程与测试结果

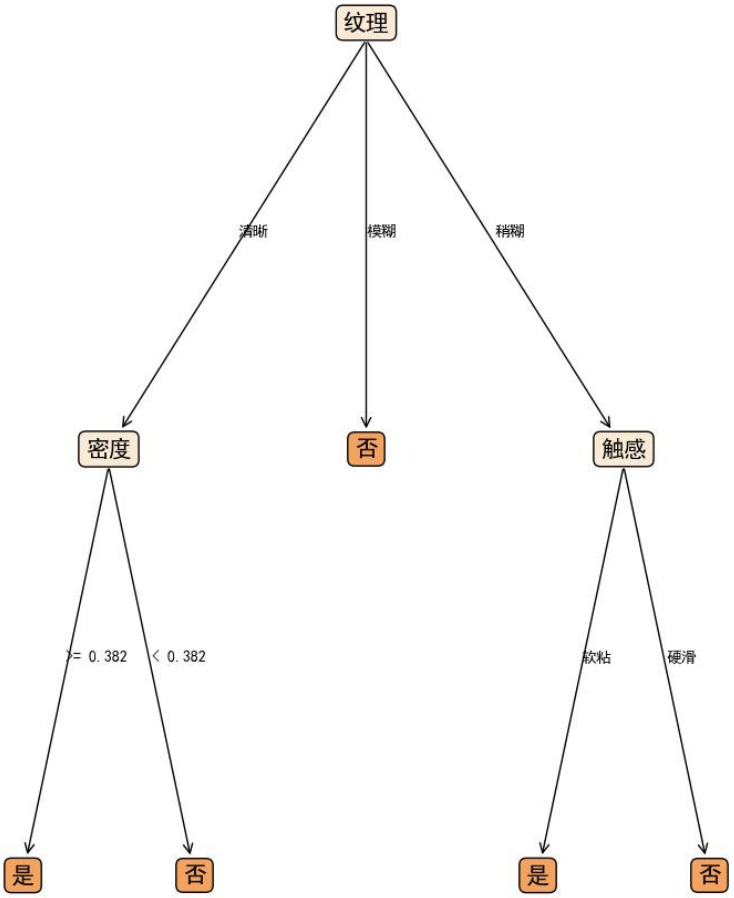
5.1.1 实验流程

首先决定算法的划分方式，是信息增益还是信息增益率。然后设定训练集与测试集的划分比例（这里我只采用了两种，0.25 与 0.2），最后进行训练，得到训练后生成的决策树图以及测试集的正确率。

5.1.2 测试结果

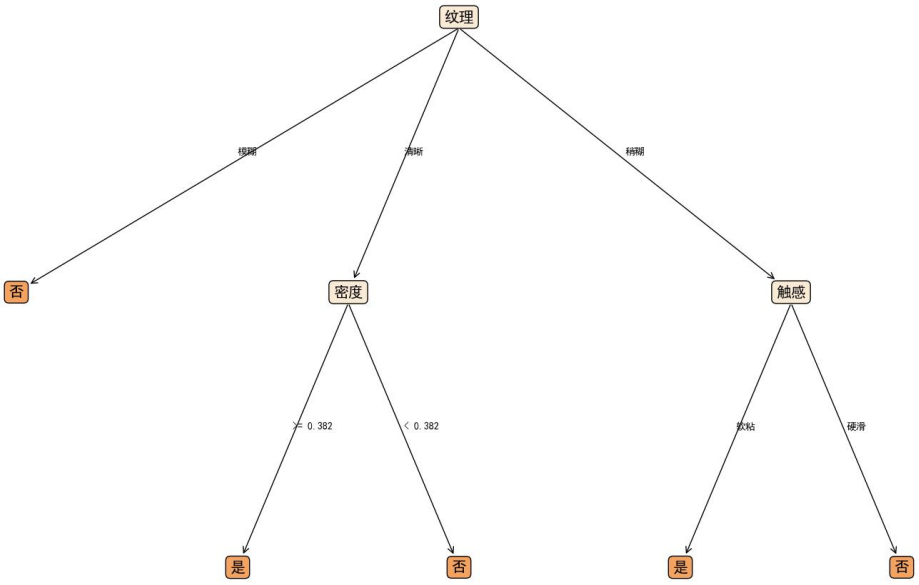
（1）划分方式信息增益率，训练集、测试集划分比例：0.2，准确率：1.0

下图为上述条件下训练得到的决策树：



（2）划分方式：信息增益率，训练集测试集划分比例：0.25，准确率：1.0

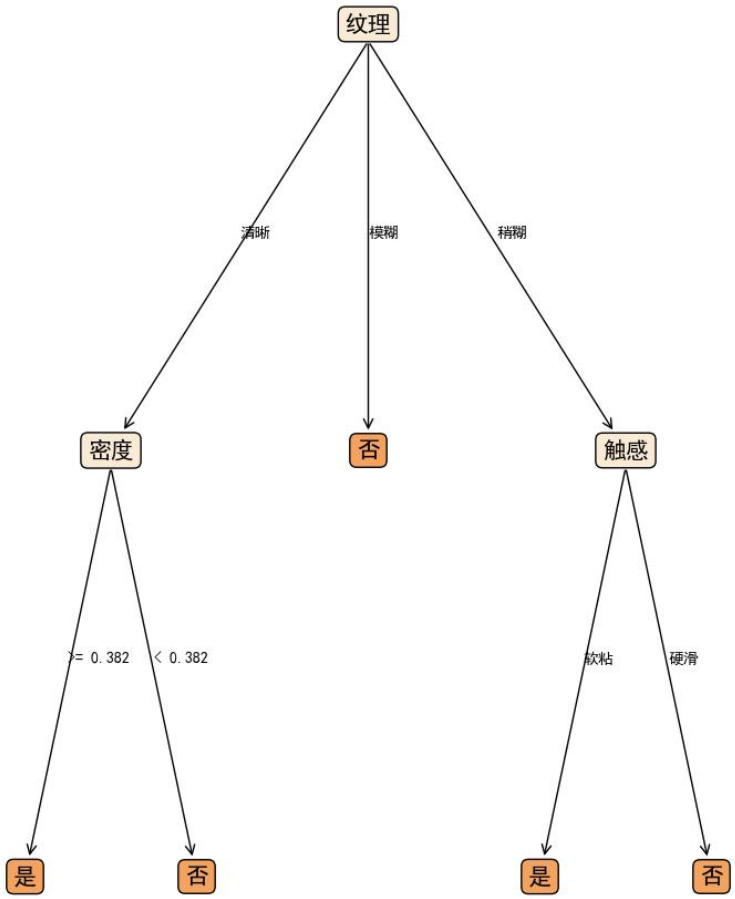
上述条件下训练得到的决策树如下：



仔细观察可以发现，划分比例为 0.25 得到的决策树与 0.2 训练得到的决策树完全一致，且准确率也为 1.0，考虑到可能是数据集太小，测试不够充分。

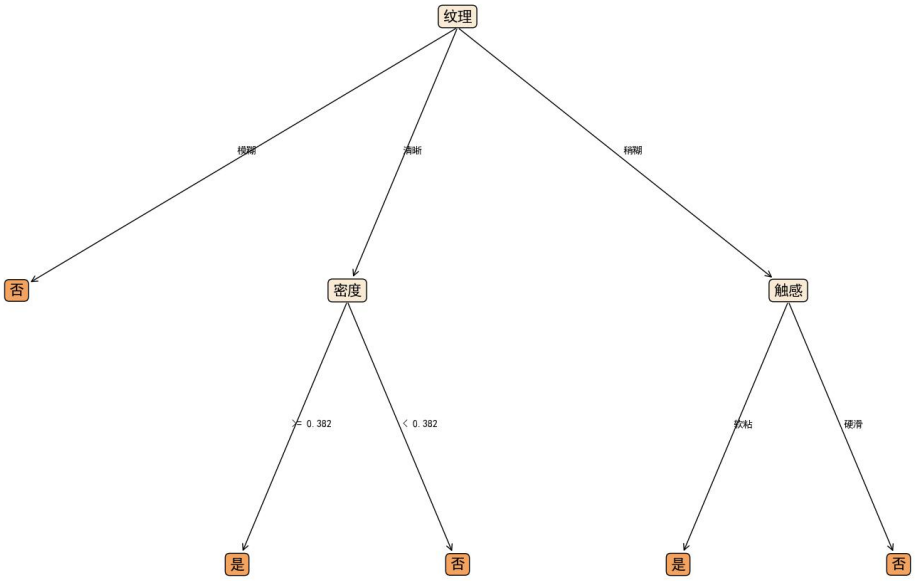
（3）划分方式：信息增益；训练集测试集划分比例：0.25；训练准确率：1.0

训练得到的决策树如下图：



（4）划分方式：信息增益；训练集测试集比例：0.25；准确率：1.0

训练得到的决策树如下图：



5.2 结果分析

本次实验中，我采用了信息增益以及信息增益率两种划分方式，并且使用了两种训练集测试集划分比例，分别为 0.2 与 0.25，进行了 4 次训练。但得到的四次结果均相同，准确率均为 1。

考虑到西瓜数据集 3.0 是一个相对较小的数据集，特征数量有限。如果数据集的规模较小，且特征之间的区分度较高，决策树可能很容易就达到 100%的准确率。

训练集和测试集的划分比例分别为 0.2 和 0.25，可能导致训练集和测试集之间的差异较小，尤其是在数据集较小的情况下。可能是由于我的训练集和测试集之间的差异不大，模型在训练集上表现良好，在测试集上也可能表现良好。

至于两种划分方法所得到的结果也一致，可能是由于：信息增益和信息增益率都是用于选择特征的方法，而西瓜数据集的特征数量比较少，它们可能会选择相同的特征进行划分，从而导致生成的决策树结构一致。

六、题目 2.1 实验流程、测试结果及分析

6.1 流程与测试结果

6.1.1 数据集介绍

本次实验采用的数据集为乳腺癌数据集（Breast Cancer Wisconsin Dataset）。这是一个经典的机器学习数据集，主要用于分类任务。它包含了 569 个样本，每个样本对应于一位乳腺癌患者的肿瘤特征。数据集的特征包括 30 个不同的细胞核特征，如半径、纹理、周长、面积、光滑度等。

样本的标签有两个类别：良性（benign）和恶性（malignant）肿瘤。这个数据集广泛用于测试各种机器学习算法的性能，尤其是分类算法。由于其易用性和明确的分类目标，它在教学和研究中具有重要的地位。

6.1.2 实验流程

首先选择算法的划分方式为**信息增益**，依次选择不剪枝、预剪枝、后剪枝的剪枝方式，训练得到不同的决策树以及训练准确率，比较不同的剪枝方式带来的差异。

在进行训练之前，首先要进行数据集的划分，第一步从乳腺癌数据集中随机选择 80%的样本作为训练集，剩余的 20%样本作为测试集，用于评估模型在未见数据上的性能。在训练集中，再次随机选择 25%的样本作为验证集。这个集用于剪枝策略，避免过拟合。

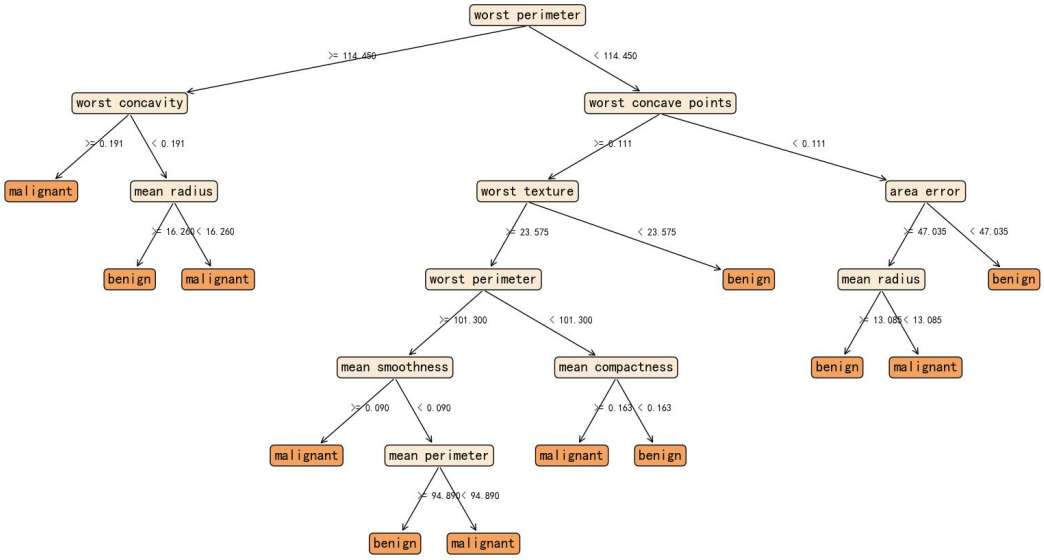
6.1.2 测试结果

三次训练的准确率分别为：

剪枝方式	准确率
不剪枝	0.9122807017543859
预剪枝	0.868421052631579
后剪枝	0.9122807017543859

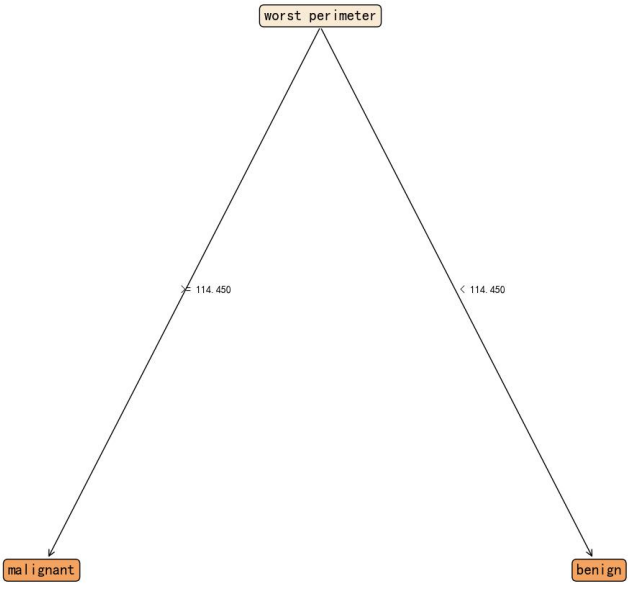
（1）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；不剪枝

下图为上述条件下训练得到的决策树：



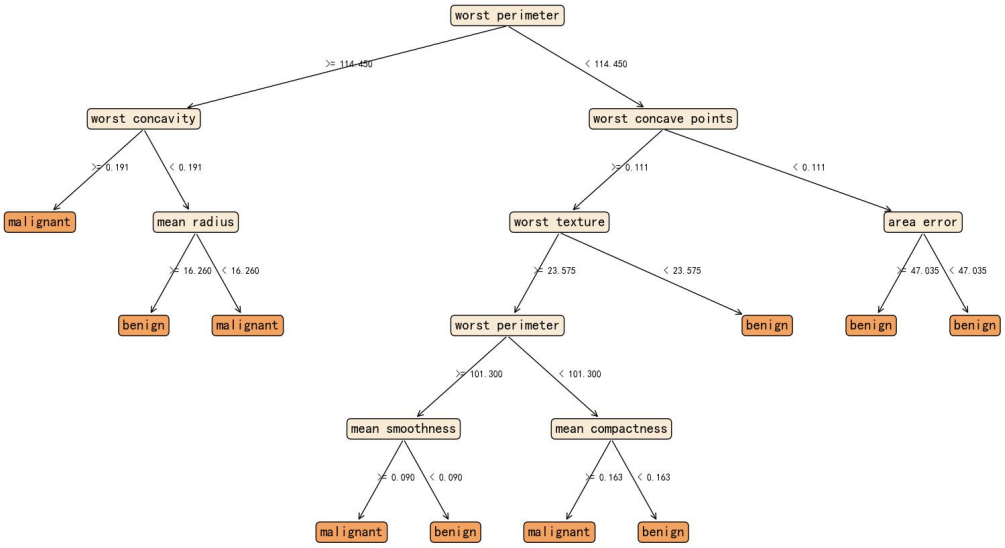
（2）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；不剪枝

下图为上述条件下训练得到的决策树：



（3）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；不剪枝

下图为上述条件下训练得到的决策树：



6.2 结果分析

对比不剪枝、预剪枝和后剪枝的决策图可以发现，预剪枝去掉了大部分节点，只剩下一个特征，大大削减了决策树的复杂程度。后剪枝的规模相对较小，可以防止模型的过拟合。

再结合三种剪枝方式的准确率可以发现，不剪枝的准确率为 0.912，说明模型在训练集上学习得非常好，能够较好地分类样本。而预剪枝准确率下降至 0.868，表明在训练过程中提前停止了一些分支，导致模型在验证集上性能下降。后剪枝的准确率恢复到 0.912，但是与剪枝前的相同。虽然没有提高准确率，但是后剪枝成功地去除了冗余的复杂性，提升了模型的泛化能力，避免了过拟合。

七、题目 2.2 实验流程、测试结果及分析

7.1 流程与测试结果

7.1.1 数据集介绍

本次实验采用的数据集为乳腺癌数据集（Breast Cancer Wisconsin Dataset）。这是一个经典的机器学习数据集，主要用于分类任务。它包含了 569 个样本，每个样本对应于一位乳腺癌患者的肿瘤特征。数据集的特征包括 30 个不同的细胞核特征，如半径、纹理、周长、面积、光滑度等。

样本的标签有两个类别：良性（benign）和恶性（malignant）肿瘤。这个数据集广泛用于测试各种机器学习算法的性能，尤其是分类算法。由于其易用性和明确的分类目标，它在教学和研究中具有重要的地位。

7.1.2 实验流程

首先选择算法的划分方式为**基尼系数（gini）**，依次选择不剪枝、预剪枝、后剪枝的剪枝方式，训练得到不同的决策树以及训练准确率，比较不同的剪枝方式带来的差异。

在进行训练之前，首先要进行数据集的划分，第一步从乳腺癌数据集中随机选择 80%的样本作为训练集，剩余的 20%样本作为测试集，用于评估模型在未见数据上的性能。在训练集中，再次随机选择 25%的样本作为验证集。这个集用于剪枝策略，避免过拟合。

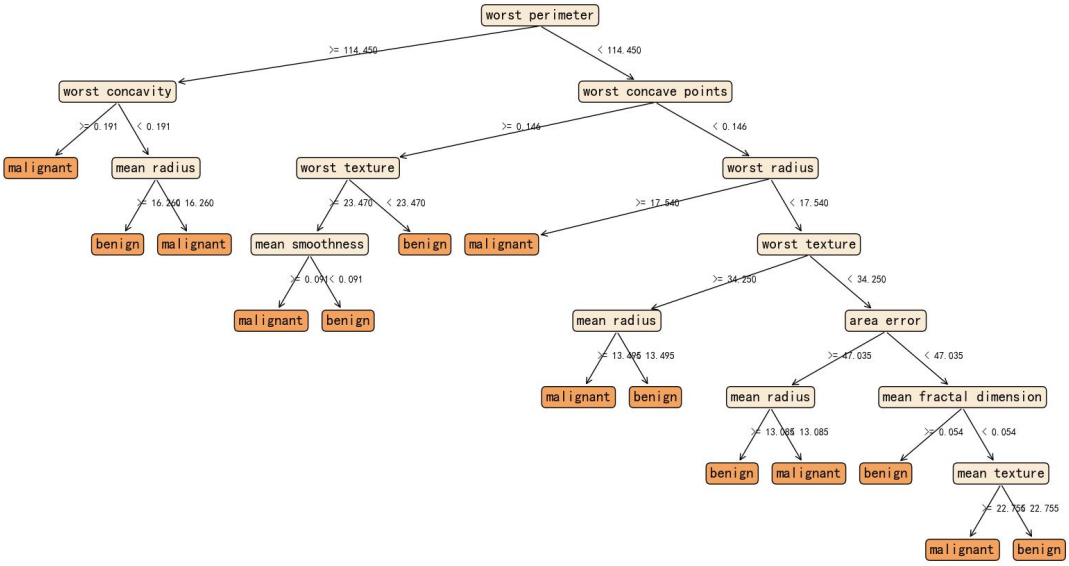
7.1.2 测试结果

三次训练的准确率分别为：

剪枝方式	准确率
不剪枝	0.9210526315789473
预剪枝	0.9210526315789473
后剪枝	0.9210526315789473

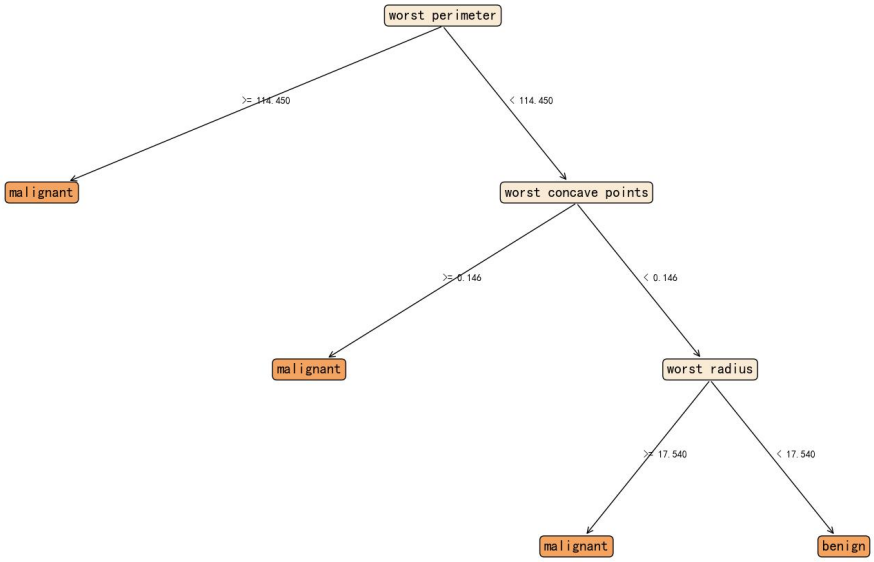
（1）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；**不剪枝**

下图为上述条件下训练得到的决策树：



（2）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；预剪枝

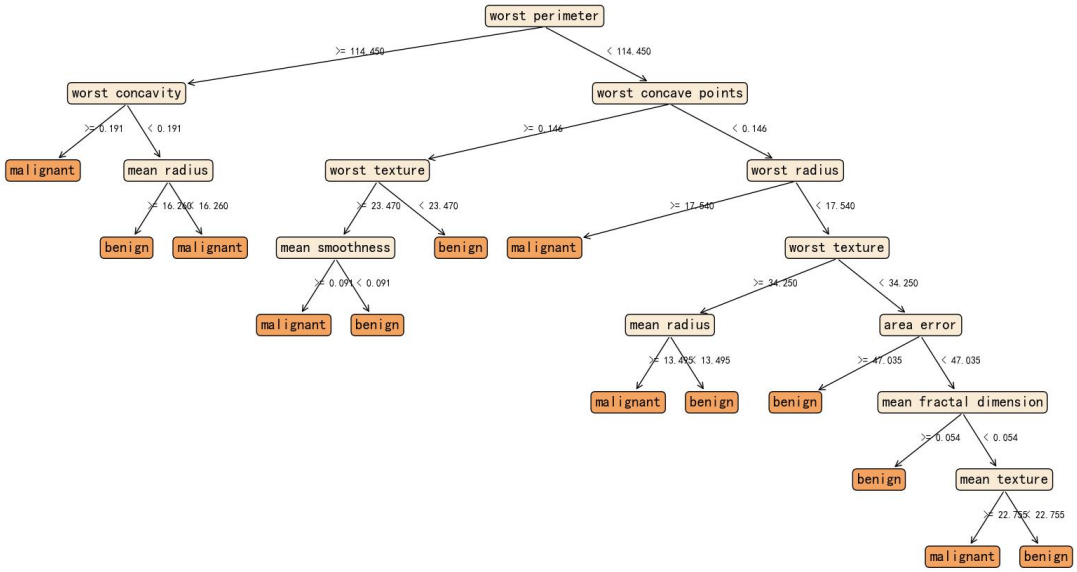
下图为上述条件下训练得到的决策树：



（3）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；后剪枝

下图为上述条件下训练得到的决策树：

、



7.2 结果分析

首先分析准确率，不剪枝的准确率为 0.921，表明模型能够很好地分类样本，具备良好的性能。而预剪枝的准确率同样为 0.921，说明预剪枝没有降低模型的准确性，这意味着在某种程度上，预剪枝没有影响模型的学习能力，能够有效地控制树的复杂度。后剪枝准确率仍然为 0.921，表明后剪枝在移除冗余节点的同时，保持了模型的性能。这表明后剪枝能够去掉不必要的复杂性，从而提高模型的泛化能力，而不会损害预测准确率。

综上所述，采用 Gini 系数时，三种剪枝方法的准确率相同，表明 Gini 系数的划分方式与决策树的结构保持了一致的性能。在这种情况下，预剪枝和后剪枝可以有效地控制模型复杂度而不影响预测能力。这种结果表明，对于乳腺癌数据集，使用 Gini 系数划分时，剪枝策略在准确性上没有显著差异，可能表明数据集的特征和目标关系非常清晰，适合多种剪枝策略。

注意到后剪枝得到大决策树图像与没有剪枝前的一致，说明后剪枝并没有实际去掉任何重要的节点。也许是不剪枝的模型没有出现明显的过拟合，后剪枝未能识别出可以去除的冗余节点，因此没有改变树的结构。而且后剪枝策略通常是在完整模型训练后进行的。如果模型在训练过程中已经较为简洁，后剪枝可能没有必要的剪去任何节点。或许在当前算法以及数据集下，剪枝的必要性比较低。

八、问题与收获

8.1 遇到的问题

首先就是特征的连续性问题，最开始我并没有考虑到特征的连续性，导致训练生成的决策树具有众多分支，训练的结果也很差，后面经过排查发现需要利用二分法对连续属性的特征济宁离散化，设定阈值，将该特征的取值分为两类。

其次就是剪枝算法的实现，一开始我没有进行正确的验证集划分，只是简单利用 sklearn 中的划分函数得到训练集以及测试集，没有单独得到验证集。这导致我的剪枝结果非常不好，后面再回看课本和查阅资料才发现是数据集划分的问题，于是进一步处理训练集，分出一部分作为剪枝时的验证集，利用验证集得到当前节点下剪枝前与剪枝后的准确率，从而更好的

进行剪枝操作。

8.2 收获

本次实验，我在实现基于信息增益（ID3）和信息增益率（C4.5）的决策树算法时，我深入理解了如何通过不同的划分方法构建决策树。这使我意识到选择合适的划分标准对树的结构和分类效果的重要性。在实验中，我学习到有效的数据集划分是提高模型性能的关键。最初，我仅仅将数据划分为训练集和测试集，没有考虑验证集的引入。这导致剪枝策略的效果不佳。通过引入验证集，我能够更好地评估剪枝前后的性能，优化模型。在处理特征的连续性问题时，我遇到了困难。初始的决策树模型因未考虑到特征的连续性而导致过度分支，影响了模型性能。经过查阅资料，我了解到使用二分法对连续特征进行离散化，可以有效简化模型结构，提高分类准确性。

通过实现未剪枝、预剪枝和后剪枝三种策略，我观察到了不同剪枝策略对模型性能的影响。尤其是后剪枝策略，它能有效去除冗余节点，保持决策树的简洁性，同时提高模型的泛化能力。这一过程让我深刻理解了剪枝的必要性和效果。

通过多次实验并调整划分比例和训练参数，我获得了不同的决策树结果，并进行性能评价。这一实践让我体会到实验的重要性，以及如何通过实验来验证和调整算法的有效性。

在整个实验过程中，我认识到在实现算法时，细节问题如数据处理、验证集的划分、特征选择等都可能影响最终结果。未来，我会更加注意这些细节，进一步提升我的数据处理能力和算法实现水平。

九、参考资料

[1]《西瓜书》
[2]决策树原理详解（无基础的同样可以看懂）-CSDN 博客
[3]【决策树】深入浅出讲解决策树算法（原理、构建） 决策树算法原理-CSDN 博客
[4]决策树原理解析 | Boole Flow
[5]决策树（decision tree）(二)——剪枝 dcp-trees-CSDN 博客
[6]决策树的划分依据之：信息增益
[7]sklearn 决策树可视化以及输出决策树规则 决策树输出规则-CSDN 博客
[8]全面！手把手教你决策树可视化（附链接&代码） | 机器之心