**《机器学习》编程作业2**

[题目1.1，2.1，2.2，2.3 2](#_Toc10954)

[一、题目理解 2](#_Toc7164)

[1.1 题目1.1 2](#_Toc29317)

[1.2 题目2.1 2](#_Toc26810)

[1.3 题目2.2 2](#_Toc28701)

[1.4 题目2.3 2](#_Toc21917)

[二、算法原理阐述 2](#_Toc17684)

[2.1 决策树算法原理 2](#_Toc7673)

[2.2 信息增益（ID3） 3](#_Toc18401)

[2.3 信息增益率（C4.5） 4](#_Toc27920)

[2.4 基尼系数 5](#_Toc4797)

[2.5 预剪枝 6](#_Toc10291)

[2.6 后剪枝 6](#_Toc16843)

[2.7 二分法处理连续值 7](#_Toc24260)

[2.8 不使用递归的决策树生成算法 7](#_Toc9093)

[三、算法设计思路 9](#_Toc14193)

[3.1 三个题目整体的实现思路 9](#_Toc30102)

[3.2 决策树类的实现思路 9](#_Toc11295)

[3.3 不使用递归的建树算法的实现思路 10](#_Toc22923)

[3.4 剪枝部分的实现思路 10](#_Toc12897)

[四、代码结构及核心部分介绍 10](#_Toc18432)

[4.1 整体代码结构 10](#_Toc7279)

[4.2 核心部分介绍 18](#_Toc15920)

[五、题目1.1实验流程、测试结果及分析 31](#_Toc7921)

[5.1 流程与测试结果 31](#_Toc277)

[5.2 结果分析 33](#_Toc2595)

[六、 题目2.1实验流程、测试结果及分析 33](#_Toc27258)

[6.1 流程与测试结果 33](#_Toc31004)

[6.2 结果分析 35](#_Toc6063)

[七、题目2.2实验流程、测试结果及分析 35](#_Toc4155)

[7.1 流程与测试结果 35](#_Toc20155)

[7.2 结果分析 37](#_Toc27652)

[八、问题与收获 37](#_Toc22078)

[8.1 遇到的问题 37](#_Toc11261)

[8.2 收获 38](#_Toc7533)

[九、 参考资料 38](#_Toc8514)

# 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：

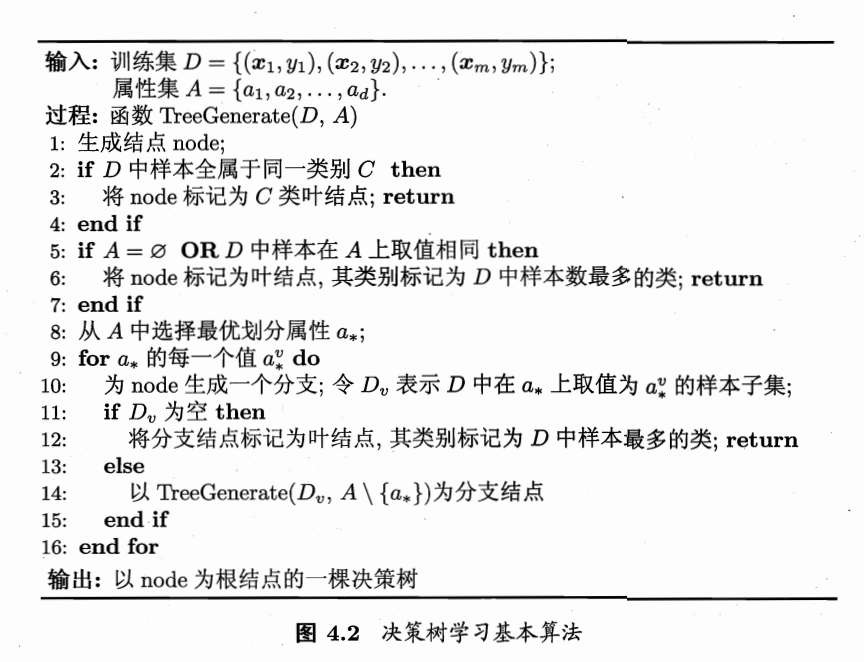


图 1 决策树学习基本算法

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。

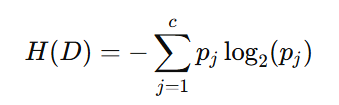
（4）样本数过少：如果当前节点的样本数少于某个预设的阈值，则停止递归。

### 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）**

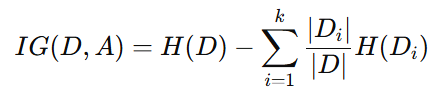
熵（Entropy）是用于衡量信息不确定性的度量。对于数据集 D，其熵 H(D)定义为：



其中，是类别的概率，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：

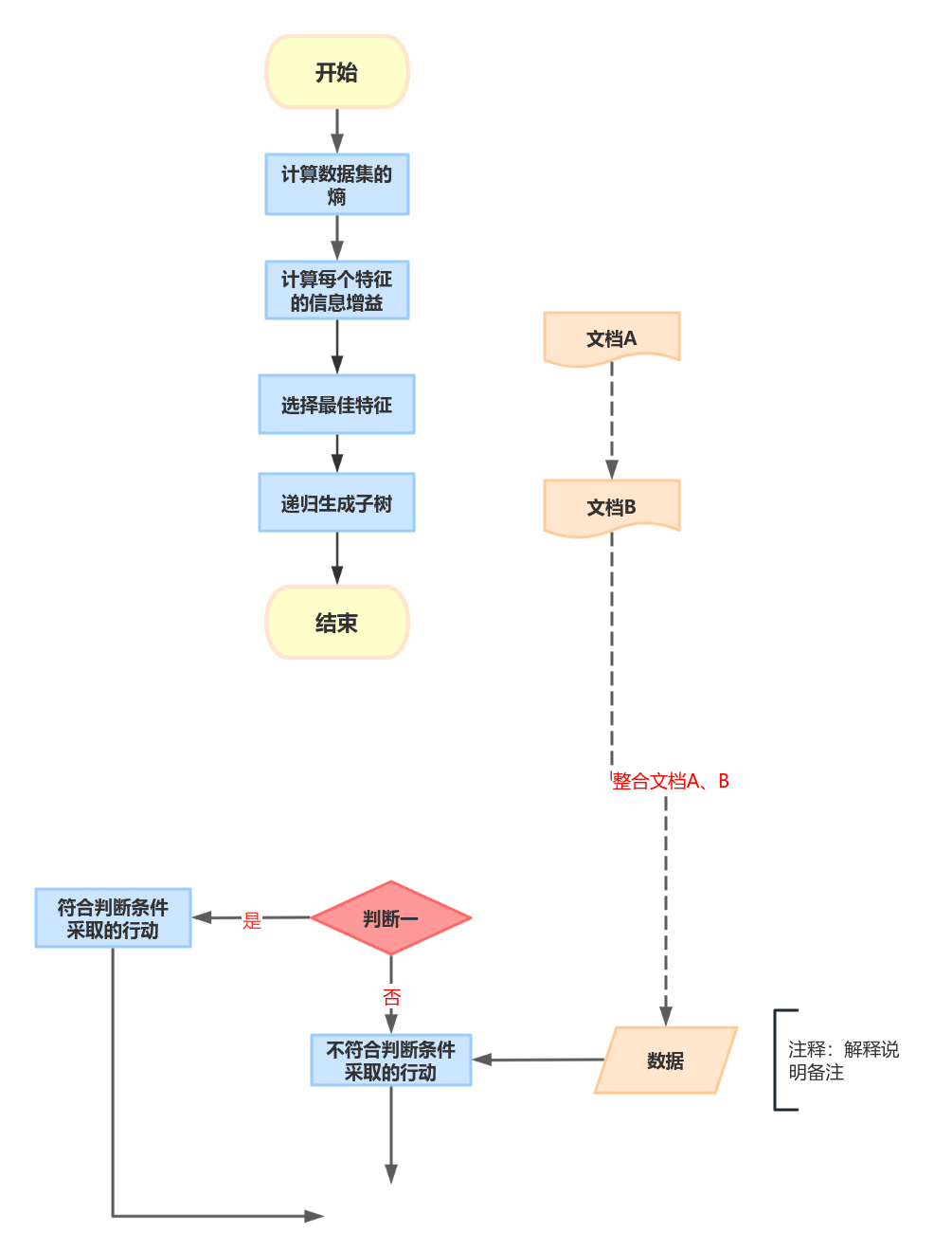


这里，∣Di∣是子集Di的样本数量，∣D∣是数据集 D的样本总数。信息增益越大，表示使用特征A划分数据集后的不确定性减少越显著。

**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。

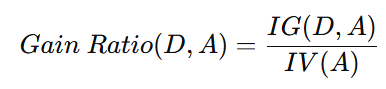
### 2.3 信息增益率（C4.5）

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。

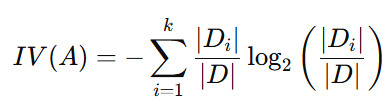
**1. 信息增益率的定义**

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（Intrinsi

c 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：



其中，IV(A)表示特征A的固有信息，定义为：



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：

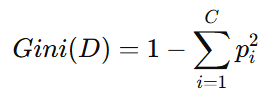
1. 计算每个特征的信息增益和固有信息。
2. 计算每个特征的信息增益率。
3. 选择信息增益率最大的特征进行划分。
4. 对每个子集递归执行上述步骤，构建决策树。
5. 执行剪枝以提高模型的泛化能力。

### 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：

​

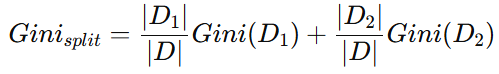
其中，pi是数据集中属于类别i的样本比例，C是类别的总数。基尼不纯度越小，表示数据集越纯，即同一类别的样本越集中。

1. **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。

1. **划分数据集**

假设特征 X将数据集 划分为两个子集 和 ，则划分的基尼不纯度计算为：



其中，|| 是数据集中样本的总数，|| 和 || 是划分后子集的样本数。

1. **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 ，然后计算合并后的节点（父节点）的预测准确率 。

**（4）决策剪枝**：如果合并后的准确率不低于子树的准确率，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**

**图示

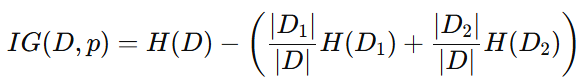
描述已自动生成**

**（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



**（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：



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={(,),(,),…,(,)}; 属性集合 A={,,…,}。  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。  11:   **for** 的每个取值 **do**  12:    生成一个子节点 child\_node，对应取值为的样本子集。  13:    **if**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,,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。

1. 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 的样本（记作）分配到 child\_node，并继续构建树。

如果为空，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。

1. 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 信息增益的实现思路

首先，定义一个函数 info\_gain，用于计算给定特征的信息增益。这个函数接收四个参数：当前特征的所有样本值、对应的标签值、当前数据集的熵，以及一个布尔值，指示特征是否为连续值。函数的主要任务是计算信息增益，并返回信息增益值。对于连续值特征，还会返回最佳分割点。

在函数内部，首先计算样本总数，然后获取特征的所有唯一值。如果是连续值特征，需要对特征值进行排序，并在每两个相邻特征值之间取中点，生成可能的分割点集合。接着，遍历所有分割点，计算每个分割点下的条件熵，并选择条件熵最小的分割点。最后，使用公式 gain = entD - min\_ent 计算信息增益，并返回信息增益和最佳分割点。

如果是离散值特征，初始化特征熵为 0，然后遍历特征的所有唯一值，计算每个取值下的条件熵，并累加到特征熵中。接着，我们使用公式 gain = entD - feature\_ent 计算信息增益，并返回信息增益。

接下来，定义另一个函数 choose\_best\_feature\_infogain，用于根据信息增益选择最佳分裂特征。这个函数接收两个参数：当前所有特征的数据和标签值。函数的主要任务是遍历所有特征，计算每个特征的信息增益，并选择信息增益最大的特征。

在函数内部，首先获取所有特征的名称，并初始化最佳特征和信息增益为 None 和负无穷。然后，计算当前数据集的熵。接着，遍历所有特征，判断特征是否为连续值，并调用 info\_gain 函数计算当前特征的信息增益。如果当前特征的信息增益大于最佳信息增益，就更新最佳特征和信息增益。最后，返回最佳特征的名称和信息增益值。

通过这两个函数的实现，决策树可以根据信息增益准则选择最佳的分裂特征，从而构建出更加有效的决策树模型。

### 3.4 基尼系数的实现思路

首先，定义一个函数 gini\_index，用于计算给定特征的基尼指数。这个函数接收三个参数：当前特征的所有样本值、对应的标签值，以及一个布尔值，指示特征是否为连续值。函数的主要任务是计算基尼指数，并返回基尼指数值。对于连续值特征，还会返回最佳分割点。

在函数内部，首先计算样本总数，然后获取特征的所有唯一值。如果是连续值特征，需要对特征值进行排序，并在每两个相邻特征值之间取中点，生成可能的分割点集合。接着，遍历所有分割点，计算每个分割点下的基尼指数，并选择基尼指数最小的分割点。最后，返回基尼指数和最佳分割点。

如果是离散值特征，初始化基尼指数为 0，然后遍历特征的所有唯一值，计算每个取值下的基尼值，并累加到基尼指数中。接着，返回基尼指数。

接下来，定义另一个函数 choose\_best\_feature\_gini，用于根据基尼指数选择最佳分裂特征。这个函数接收两个参数：当前所有特征的数据和标签值。函数的主要任务是遍历所有特征，计算每个特征的基尼指数，并选择基尼指数最小的特征。

在函数内部，首先获取所有特征的名称，并初始化最佳特征和基尼指数为 None 和正无穷。然后，遍历所有特征，判断特征是否为连续值，并调用 gini\_index 函数计算当前特征的基尼指数。如果当前特征的基尼指数小于最佳基尼指数，就更新最佳特征和基尼指数。最后，返回最佳特征的名称和基尼指数值。

类似于信息增益，通过这两个函数的实现，决策树可以根据基尼系数准则选择最佳的分裂特征，从而构建出更加有效的决策树模型。

### 3.5 信息增益率的实现思路

首先，定义一个函数 info\_gainRatio，用于计算给定特征的信息增益率。这个函数接收四个参数：当前特征的所有样本值、对应的标签值、当前数据集的熵，以及一个布尔值，指示特征是否为连续值。函数的主要任务是计算信息增益率，并返回信息增益率值。对于连续值特征，还会返回最佳分割点。

在函数内部，如果是连续值特征，首先调用 info\_gain 函数计算信息增益和最佳分割点。然后，计算小于和大于分割点的样本占比，并根据这些占比计算特征的熵（IV）。接着，对信息增益进行修正，减去 log2(N-1)/|D|，其中 N 是当前特征的取值个数，D 是总数据量。最后，计算信息增益率，并返回信息增益率和最佳分割点。

如果是离散值特征，首先计算特征各取值的样本占比，并根据这些占比计算特征的熵（IV）。然后，调用 info\_gain 函数计算信息增益，并计算信息增益率。最后，返回信息增益率。

接下来，定义另一个函数 choose\_best\_feature\_gainratio，用于根据信息增益率选择最佳分裂特征。这个函数接收两个参数：当前所有特征的数据和标签值。函数的主要任务是遍历所有特征，计算每个特征的信息增益率，并选择信息增益率最大的特征。

在函数内部，首先获取所有特征的名称，并初始化最佳特征和信息增益率为 None 和负无穷。然后，计算当前数据集的熵。接着，遍历所有特征，判断特征是否为连续值，并调用 info\_gainRatio 函数计算当前特征的信息增益率。如果当前特征的信息增益率大于最佳信息增益率，就更新最佳特征和信息增益率。最后，返回最佳特征的名称和信息增益率值。

通过这两个函数的实现，决策树可以根据信息增益率准则选择最佳的分裂特征，从而构建出更加平衡和有效的决策树模型。

### 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。

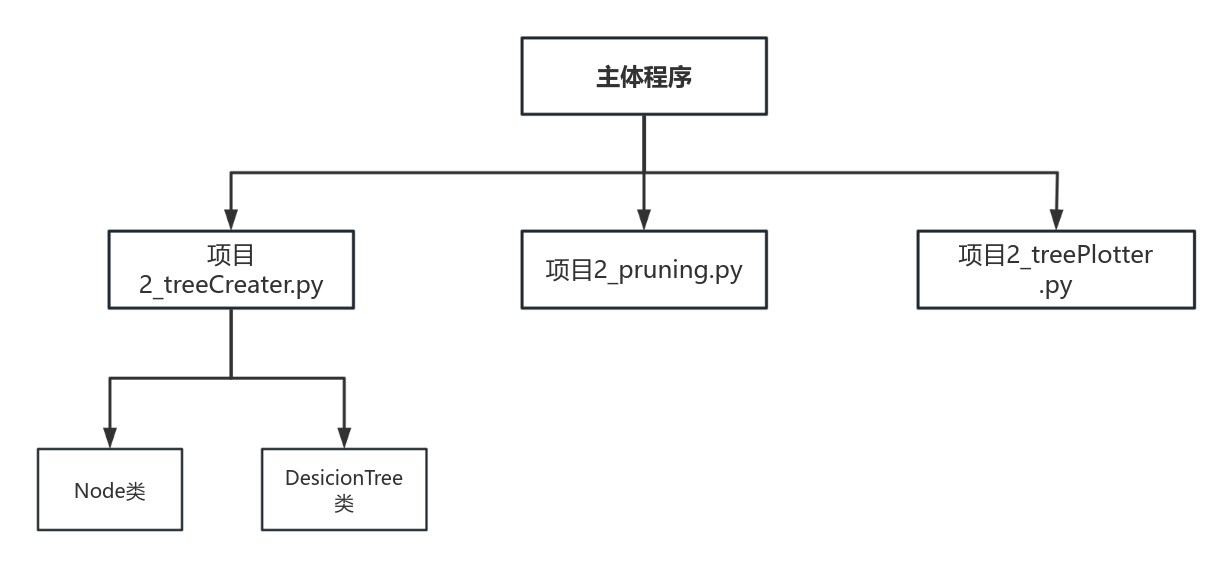
### 3.4 剪枝部分的实现思路

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。

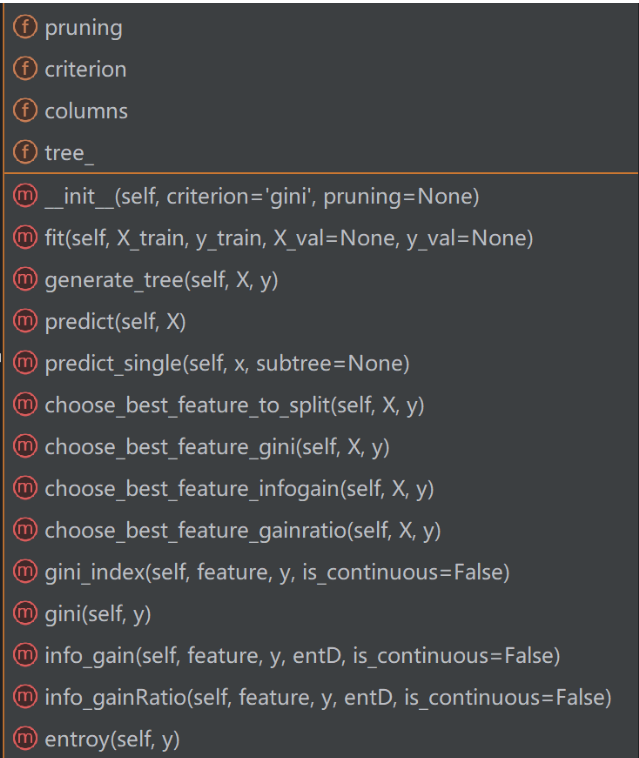
预剪枝是从根节点开始，而后剪枝是从最底层的属性节点开始计算：从树的叶子节点开始，逐步向上遍历每个节点。

## 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):**

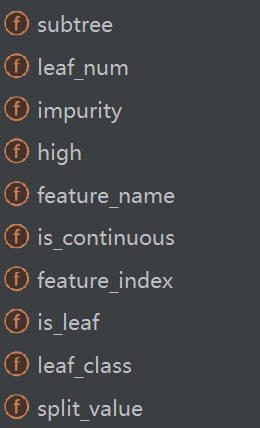
**参数:**

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 |

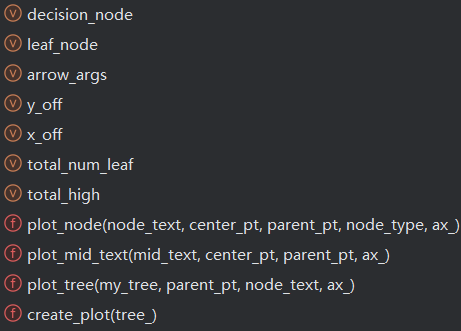
**2.Node类的成员变量：**



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。

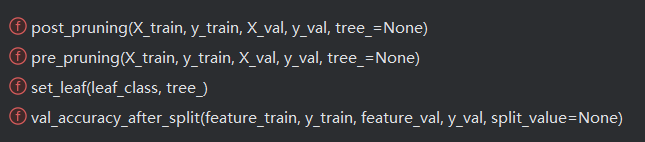
4、更新 y\_off 和 x\_off，以便在递归过程中正确放置节点。

|  |
| --- |
| 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。

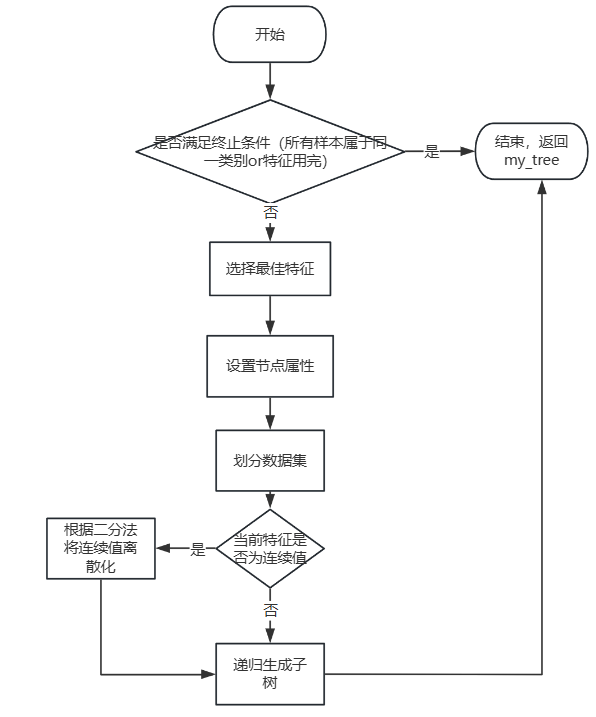


图 2 generate\_tree函数的实现思路

**（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。

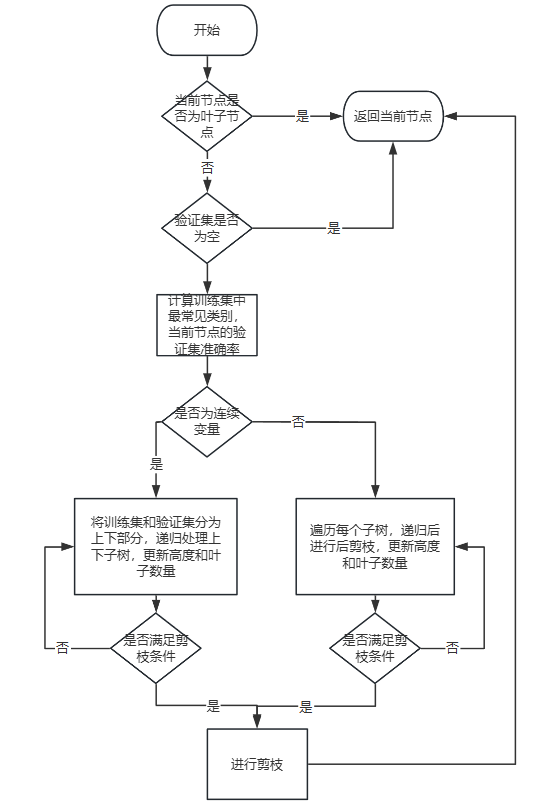
如果为连续值特征，即 is\_continuous 为 True，则对特征值进行排序，并生成所有可能的分割点。遍历所有分割点，计算每个分割点的信息熵，选择信息熵最小的分割点。然后计算信息增益 gain。并返回信息增益值gain以及连续值划分点（二分法划分）。

**（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\_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   # 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。

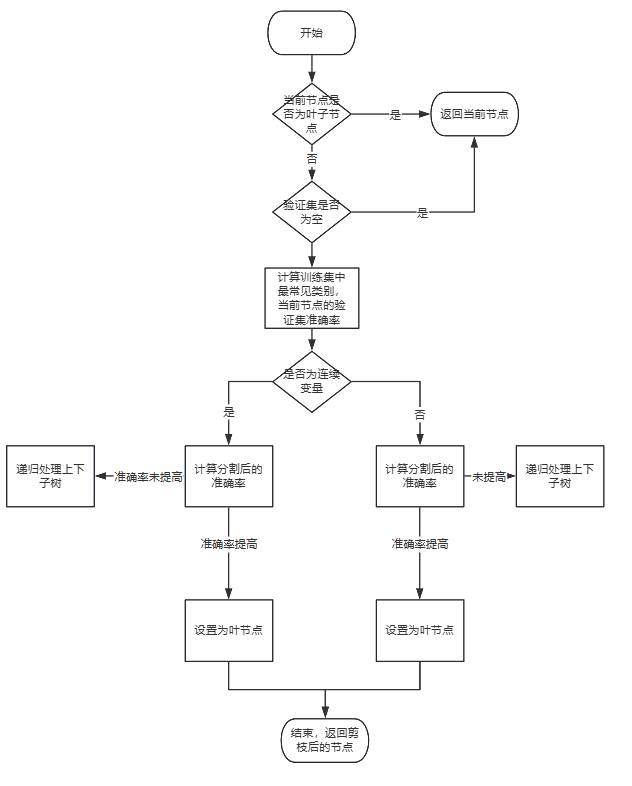
首先检查当前节点是否为叶子节点。如果是，直接返回；如果验证集为空，也不进行剪枝。

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。

1. **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\_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)  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.empty:  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, current\_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[feature\_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.split\_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

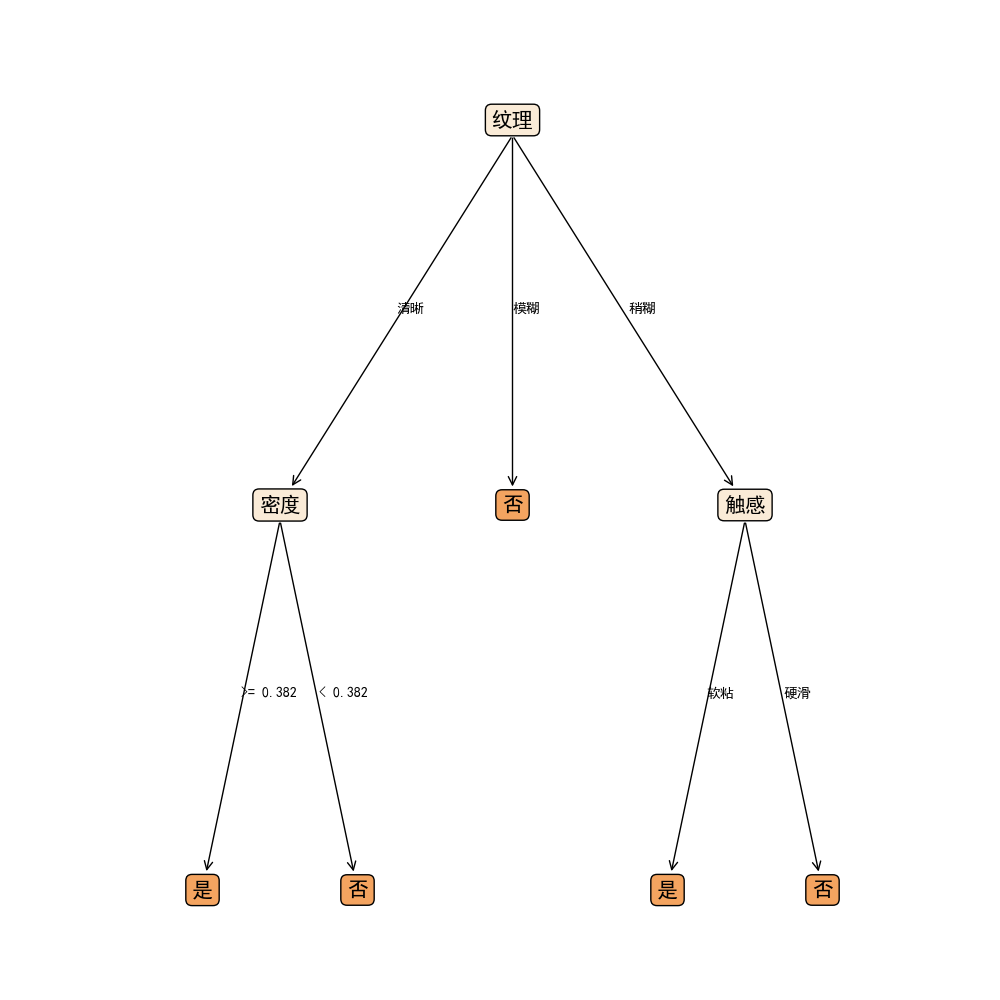
**5.1.1 实验流程**

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**

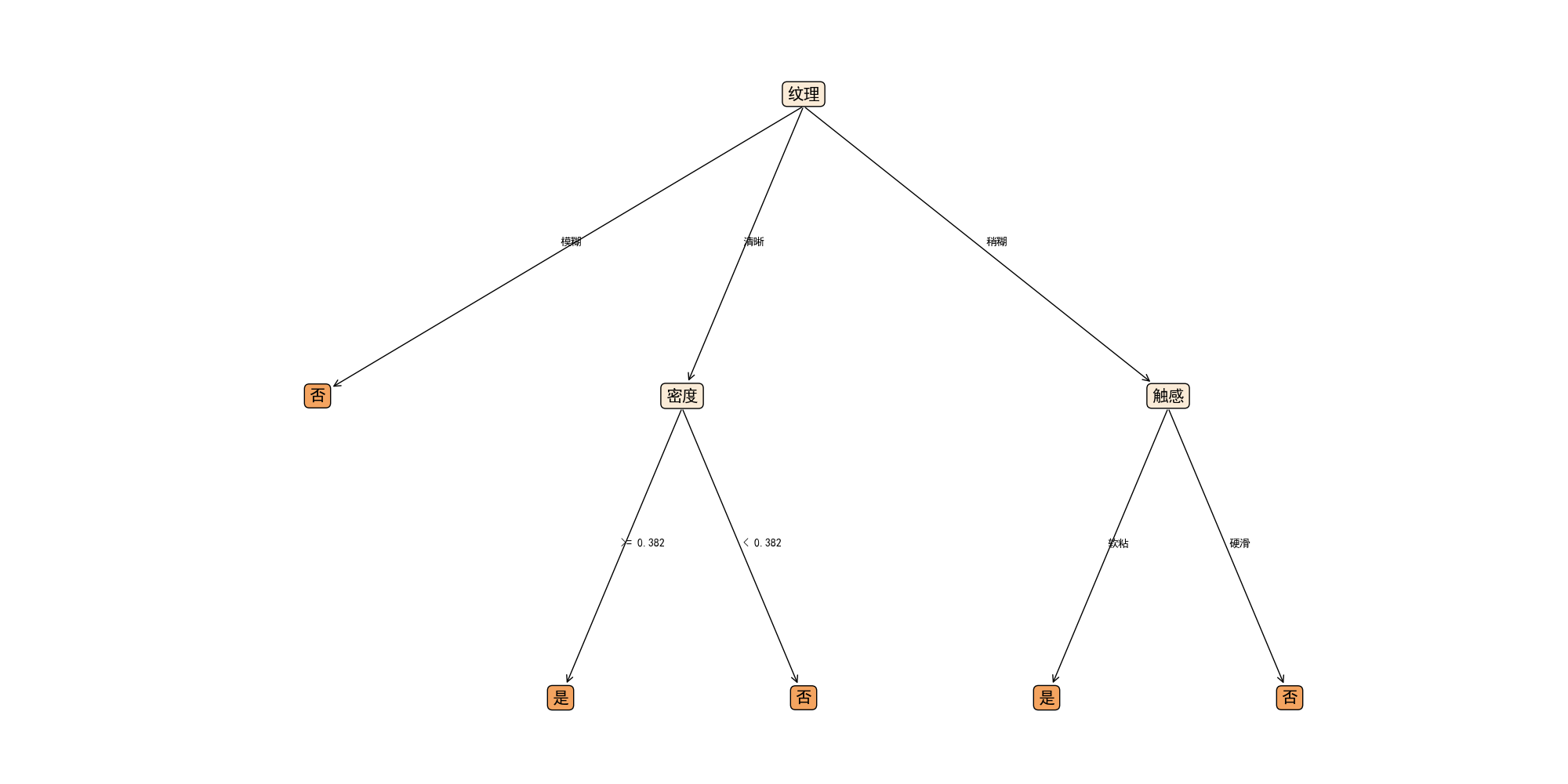
（1）划分方式信息增益率，训练集、测试集划分比例：0.2，准确率：1.0

下图为上述条件下训练得到的决策树：



1. 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

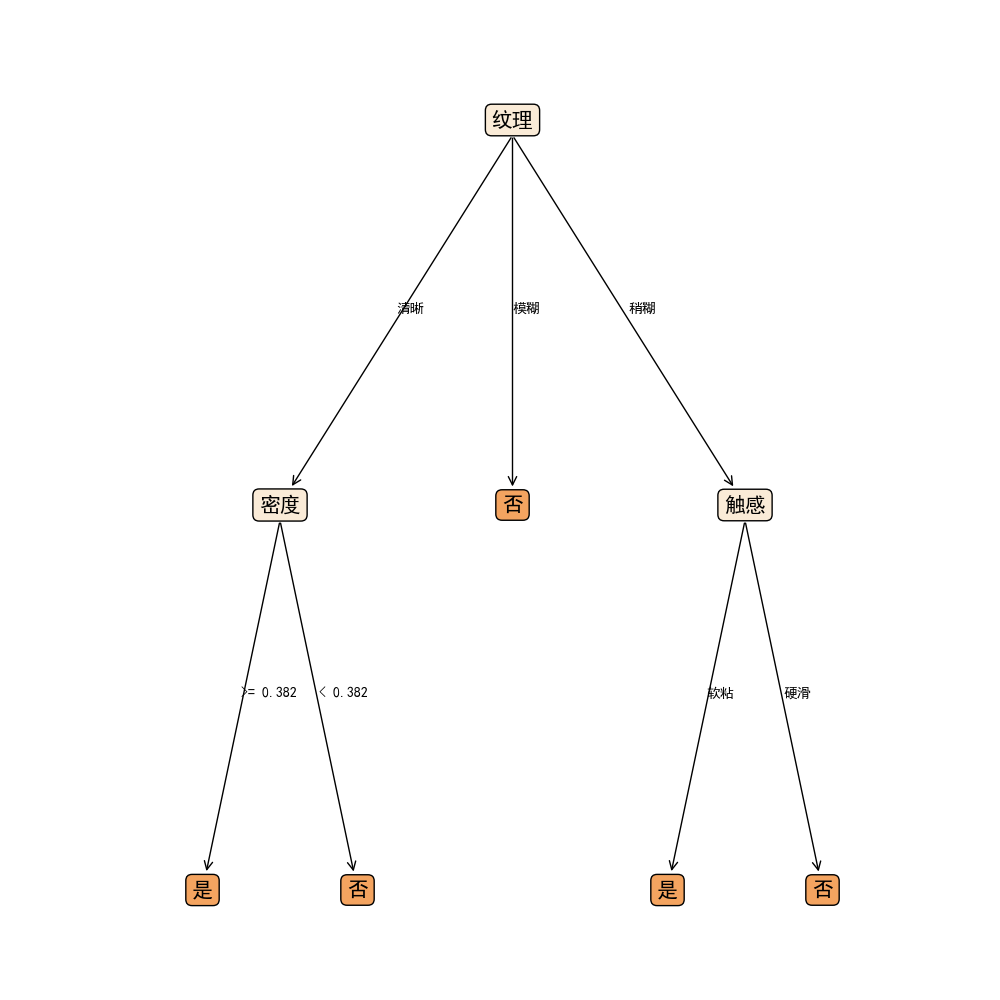
上述条件下训练得到的决策树如下：



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。

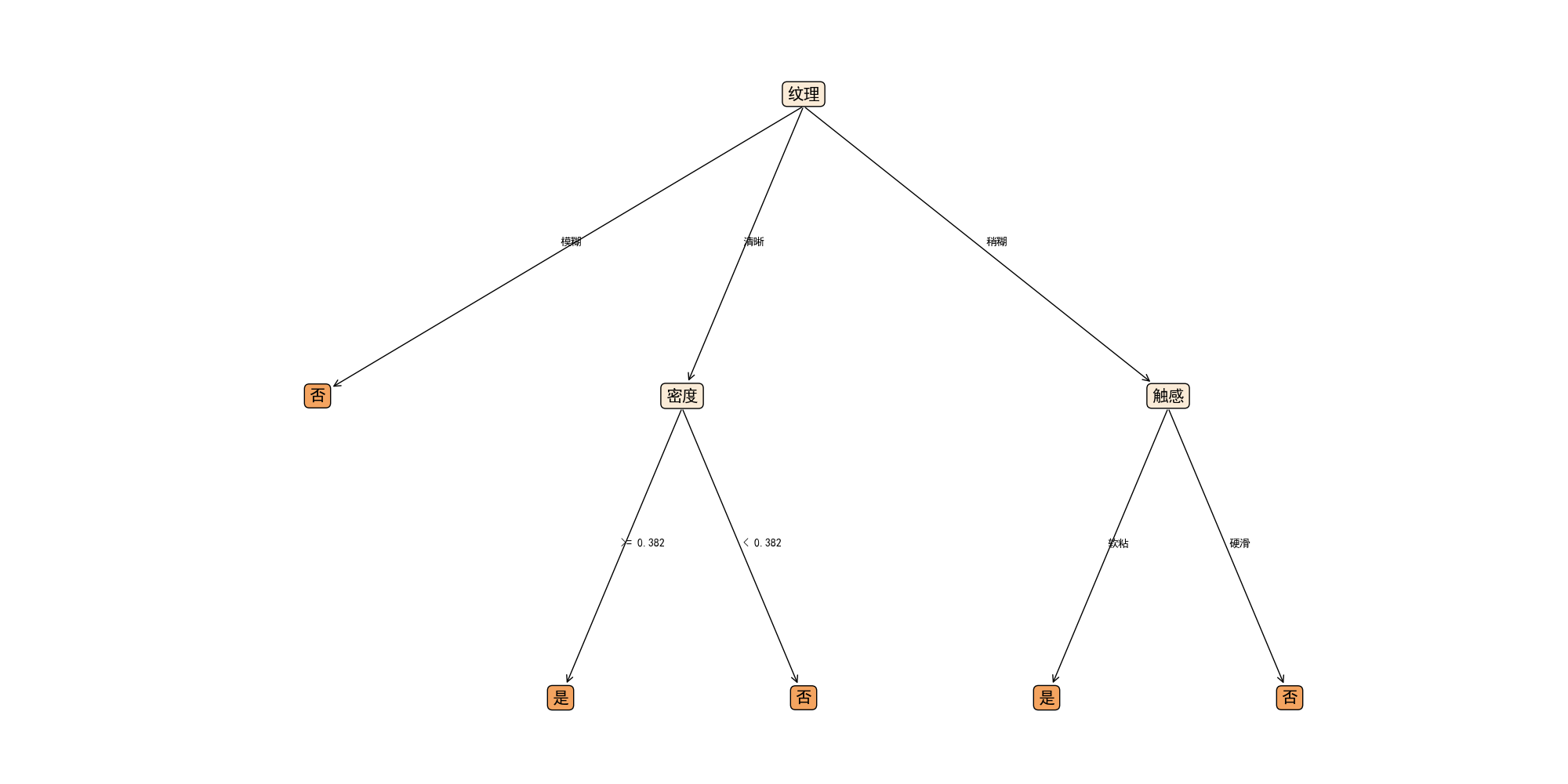
1. 划分方式：信息增益；训练集测试集划分比例：0.25；训练准确率：1.0

训练得到的决策树如下图：



1. 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。

## 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。

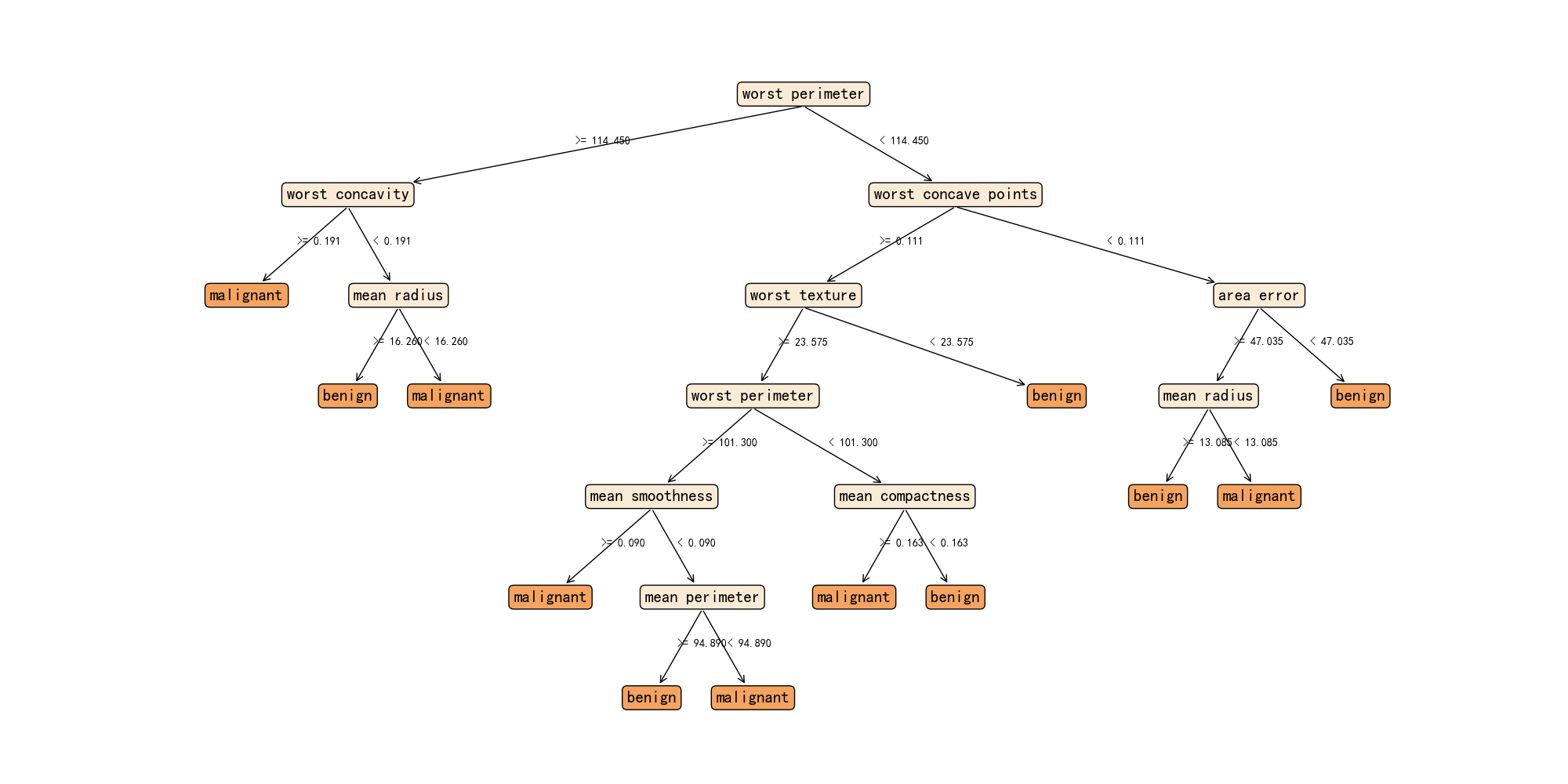
**6.1.2 测试结果**

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 |

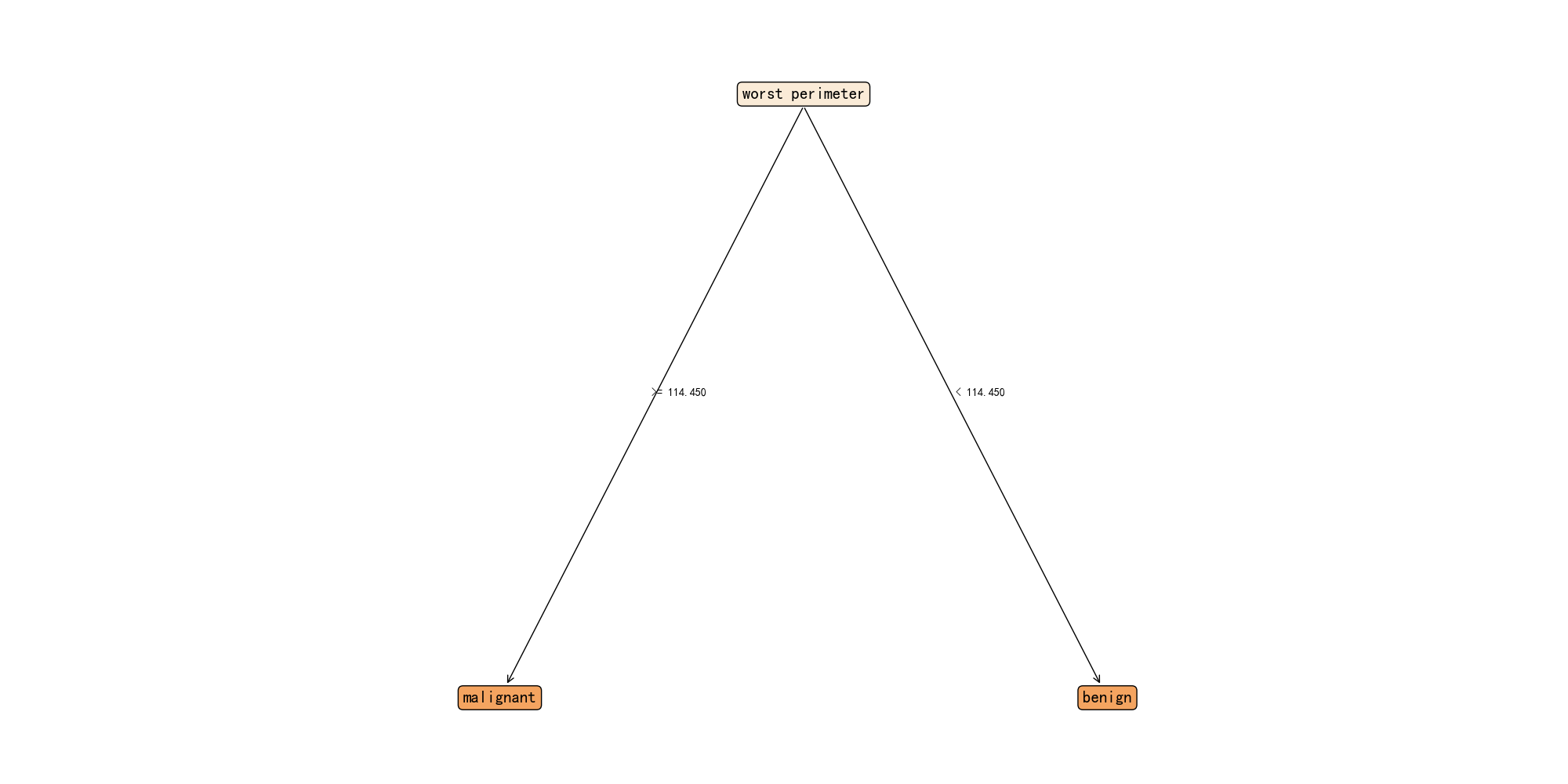
1. 训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；不剪枝

下图为上述条件下训练得到的决策树：



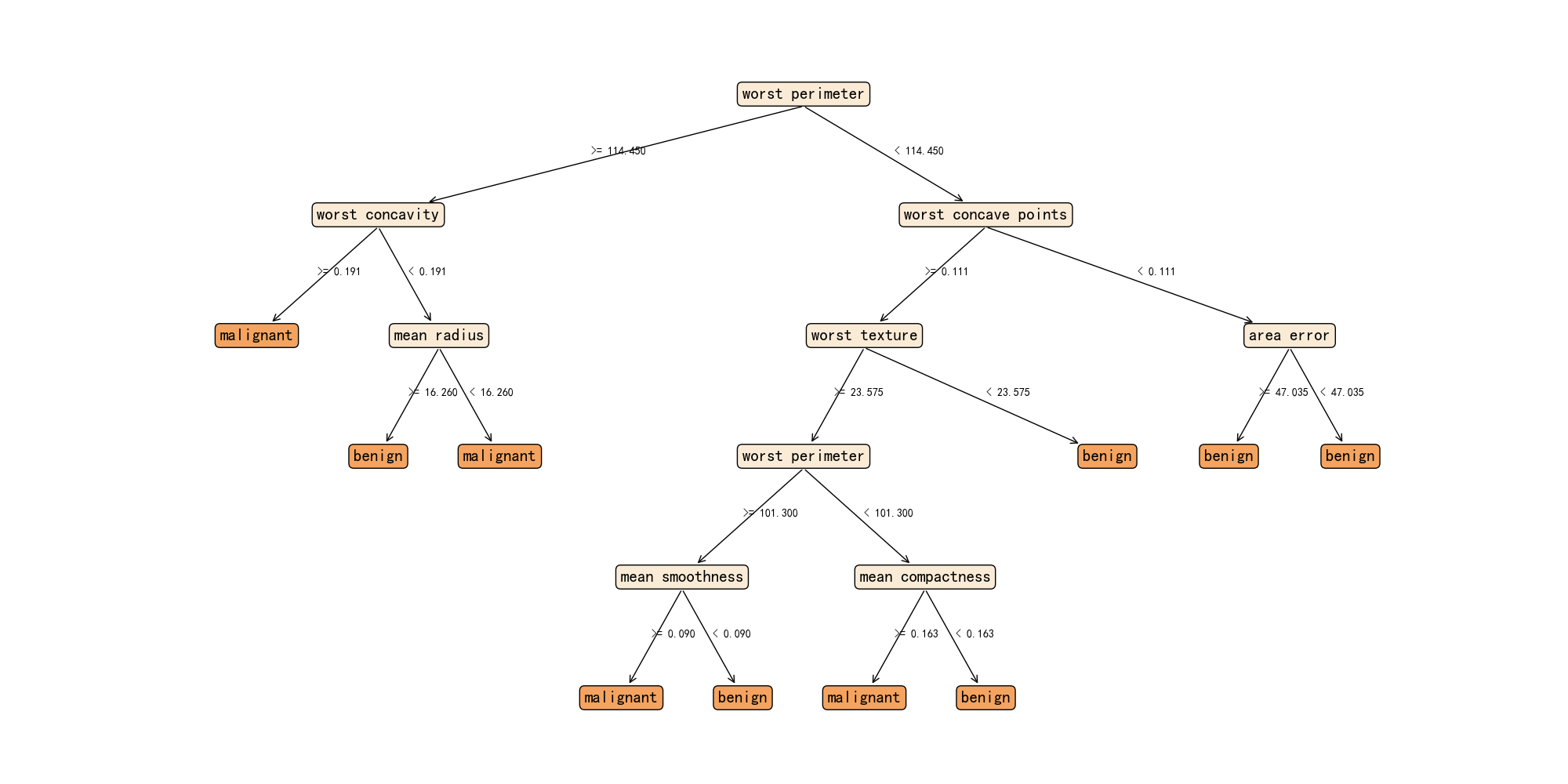
1. 训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；不剪枝

下图为上述条件下训练得到的决策树：



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：



### 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。

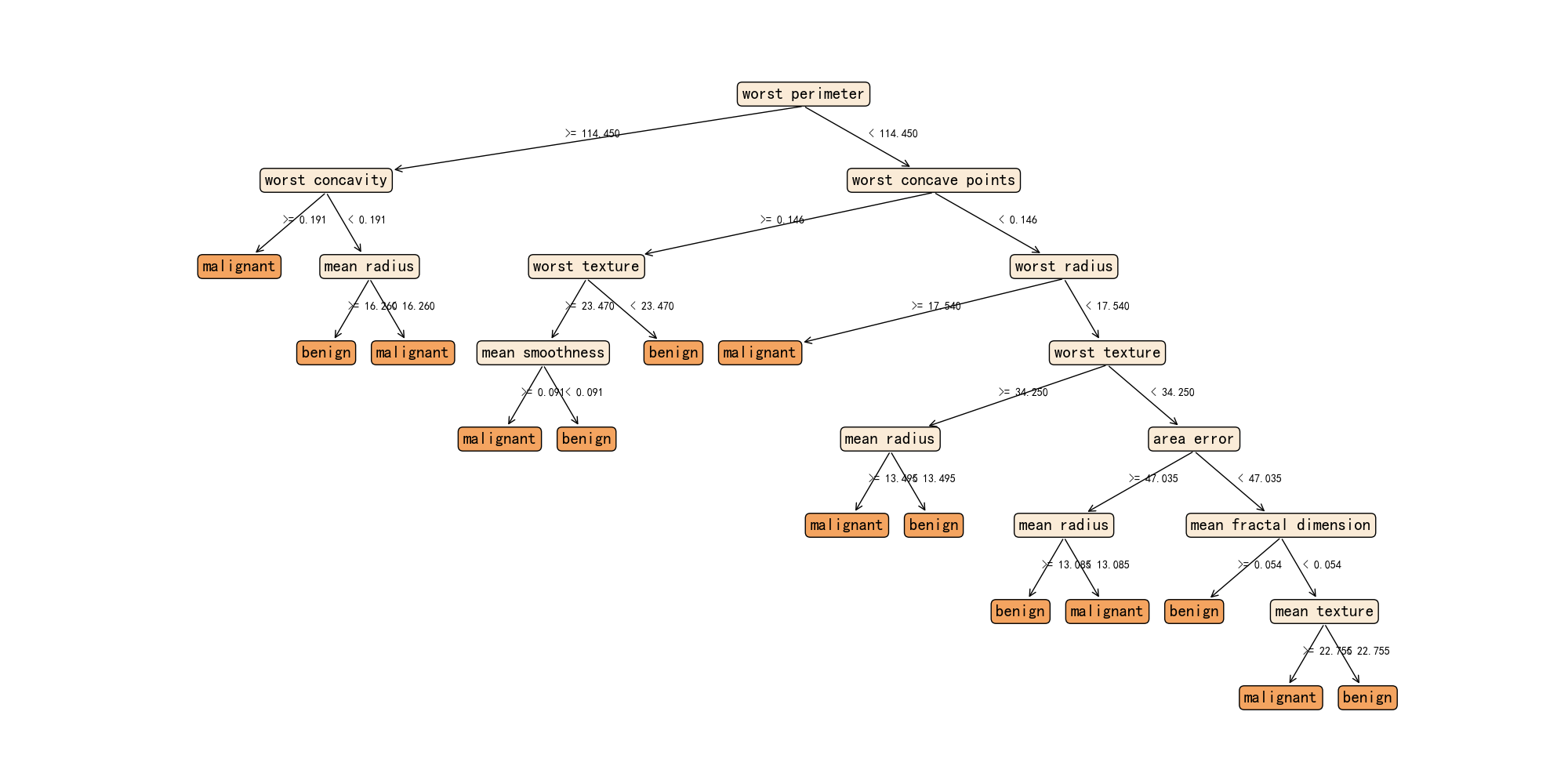
**7.1.2 测试结果**

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 |

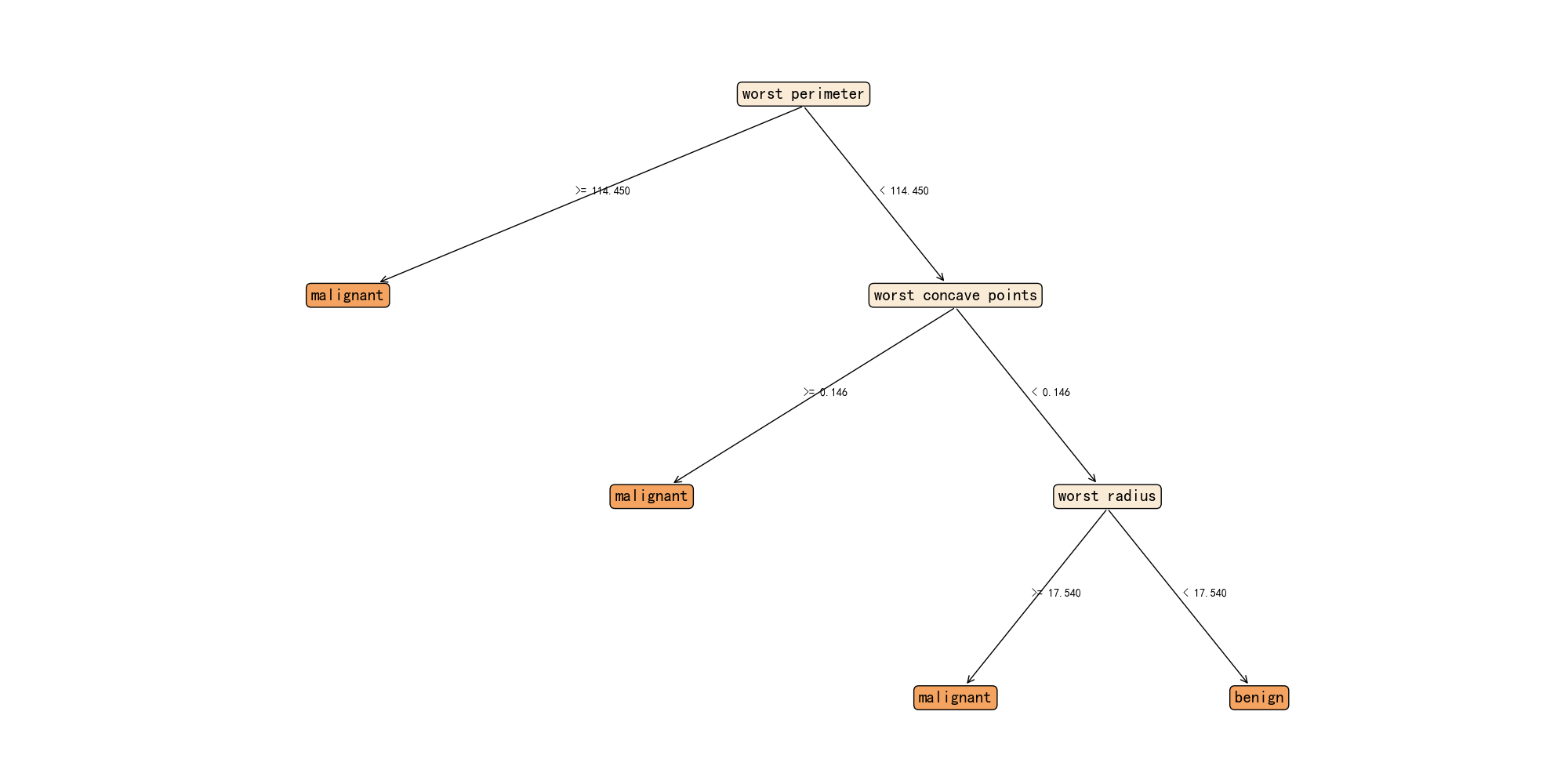
（1）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；**不剪枝**

下图为上述条件下训练得到的决策树：



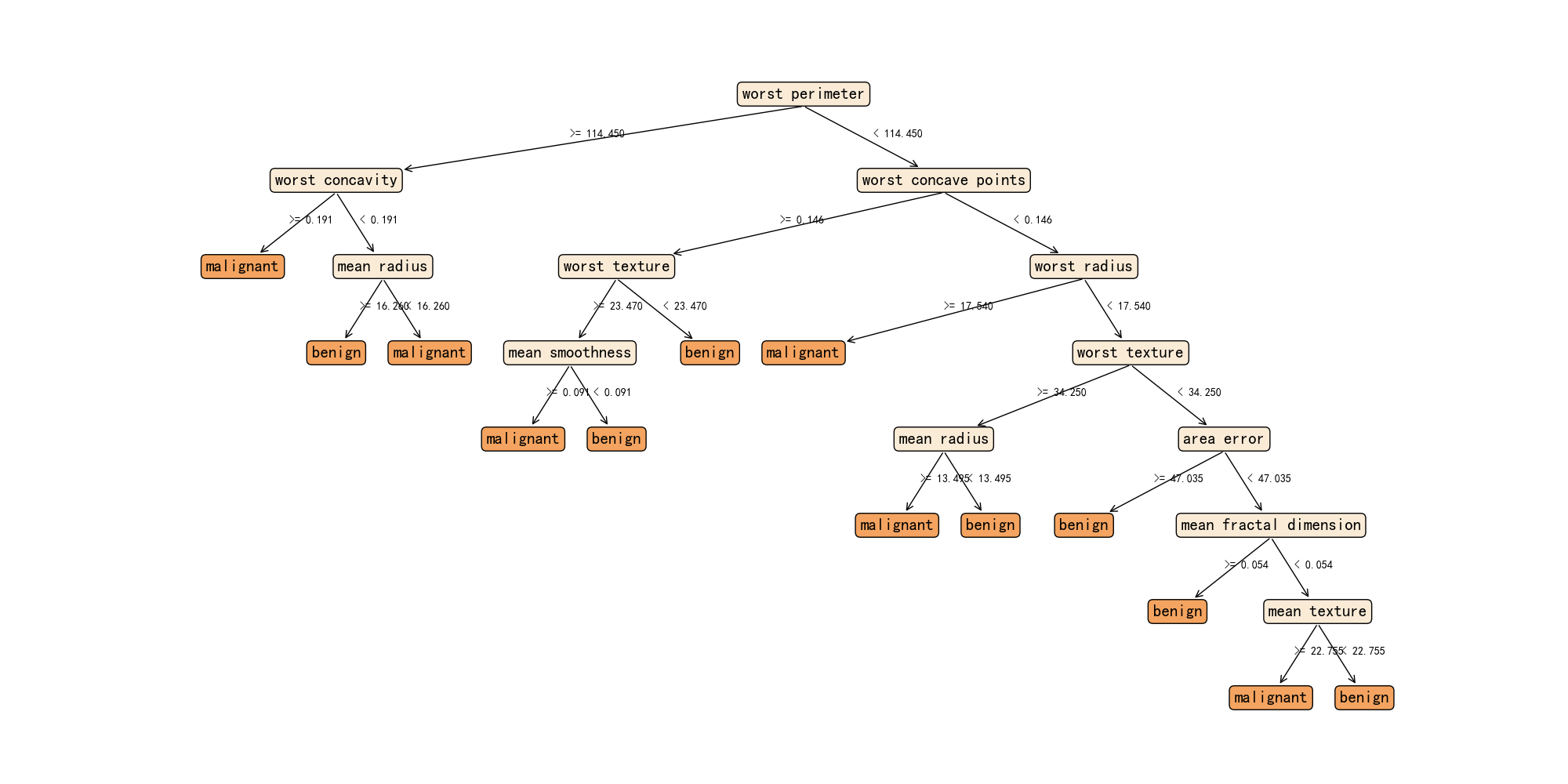
（2）训练集、测试集划分比例：0.2；训练集中剪枝验证集的划分比例：0.25；预剪枝

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。

## 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](https://blog.csdn.net/GreenYang5277/article/details/104500739)

[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](https://blog.csdn.net/kevinjin2011/article/details/125147134)

[4][决策树原理解析 | Boole Flow](https://booleflow.com/2021/01/16/jue-ce-shu-yuan-li-jie-xi/)

[5][决策树（decision tree）(二)——剪枝\_dcp-trees-CSDN博客](https://blog.csdn.net/u012328159/article/details/79285214)

[6][决策树的划分依据之：信息增益](https://www.itheima.com/news/20210916/180526.html)

[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](https://blog.csdn.net/qq_41103204/article/details/116778943)

[8][全面！手把手教你决策树可视化（附链接&代码） | 机器之心](https://www.jiqizhixin.com/articles/2020-01-31-4)