006 决策树

0 综述

在本次实验中， 我实现了ID3决策树， 实现了gain和gini作为划分选选择计算， 对于连续型属性，构造了C4.5树。实现了预剪枝算法， 分析了预剪枝算法是怎么工作的。

1 问题描述

基本要求

1. 基于 Watermelon-train1 数据集(只有离散属性)构造 ID3 决策树;
2. 基于构造的 ID3 决策树,对数据集 Watermelon-test1 进行预测,输 出分类精度;

中级要求

1. 基于 Watermelon-train2 数据集构造 C4.5或者 CART 决策树,要求 可以处理连续型属性;
2. 基于构造的决策树,对数据集 Watermelon-test2 进行预测,输出分类精度;

高级要求

1. 使用任意的剪枝算法对构造的决策树进行剪枝,观察测试集合的分类精度是否有提升,给出分析过程;

1.1 数据描述

有一个label， 是否是好瓜， 取值是[是, 否]。

有多个attri， 数据1的attris全是离散的， 每个attr有三个取值

数据2 有一个attr 密度 是连续的。

值得注意的是训练集和测试集都不大， 训练集不大意味着决策树比较小， 测试集不大意味着预测正确率比较容易受特殊值影响而达不到一个好看的值。

2 解决方法

2.1 解决思路

1. 理解决策树建立过程， 写一个建立决策树的函数， 需要注意的有两点
    1. 计算信息增益的接口留出， 方便单独实现，也方便使用不同实现的函数
    2. 对于连续值有不同的处理方法
2. 实现信息增益的计算函数， 主要是要理解公式
3. 实现剪枝函数

2.2 基本理论

2.2.1 决策树与决策树构建

决策树(decision tree)是一类常见的机器学习算法。 核心的思路是把从attris推断label看作一次决策， 然后接下来把这个决策一步步分成子决策。 以判断瓜甜不甜为例子， 我们想通过我们对attris[色泽,根蒂,敲声,纹理,密度]的了解决策这个瓜是否好瓜， 那么我们先看颜色是不是好， 再看敲声是不是好， 最终能在决策树的叶节点下一个判断--是好瓜或者不是好瓜。

形式化的定义， 一棵决策树有一些叶节点和非叶节点， 叶节点上要标明label，而非叶节点则对应一个决策attr， 依据这个attr的取值，每一个取值产生一颗对应的子树。

```
输入
训练集D={(x1,y1),(x2,y2),…(xm,ym)};
属性集A={a1,a2,…ad}.
过程：函数TreeGenerate(D,A, node)
    if D中样本全属于同一类别C:
        将node标记为C类叶结点;
        return
    end if
    if A=空集 or D中样本在A上取值相同:
        将node标记为D中样本数(当前结点)最多的类(成为叶结点);
        return
    end if
```

```
    从A中选择最优划分属性a*:
    for a* 的每个值 a‘* do
        为node生成一个分支；令Dv表示D中在a*上取值为a’*的样本子集
        if Dv为空:
            将分支结点标记为D中样本数(父结点)最多的类(成为叶结点);
            return
        else
            TreeGenerate (Dv, A{a*},child_of_node)
        end if
    end for
输出
以node为根结点的一棵决策树
```

我的决策树构建函数就是基于这个伪代码实现的

2.2.2 划分选择

决策树的关键是如何选择最优的划分属性， 一般而言， 随着决策树的进行， 我们希望每一个分支的节点越来越多的属于某一个类别（label相同）。即纯度更大。

接下来介绍两种不同的划分选择， 我在代码中两种都实现了。

2.2.2.1 信息增益和ID3决策树

信息熵是度量样本集合纯度最高的指标。 假定当前样本集合D中， 第k类样本出现的概率是pk， 那么当前信息熵定义是

$$Ent(D) = \sum_{k=1}^{|y|} p_k log_2 p_k \tag{1}$$

Ent（D）越小， D的纯度越高

由此定义信息增益， 对于离散属性a， 有V个可能的取值， 以不同的取值建立分支， 其中以a==v建立的分支D中a的取值都是v， 记作Dv, 有信息增益为

$$Gain(D,a) = Ent(D) - \sum_{v=1}^{V} \frac{D^v}{D} Ent(D^v) \tag{2}$$

信息增益越大， 意味着以a划分后的子树“纯度”越大。

即我们可以使用信息增益

```
从A中选择最优划分属性a*:
```

方法是

$$a_* = argmax Gain(D,a) \tag{3}$$

这样构建起来的决策树就是ID3决策树

2.2.2.2 基尼指数和CART决策树

使用基尼指数来度量数据D的纯度

$$Gini(D) = \sum_{k=1}^{|y|} \sum_{k \neq k'} p_k p_{k'} \tag{4}$$

$$= 1 - \sum_{k=1}^{|y|} p_k^2$$

属性a的基尼指数定义为

$$Gini\_index(D, a) = \sum_{v=1}^{V} \frac{|D^v|}{|D|} Gini(D^v) \tag{5}$$

基尼指数越小， 意味着以a划分后的子树"纯度"越大。

即我们可以使用基尼指数

> 从A中选择最优划分属性a*：

方法是

$$a_* = argmin Gini\_index(D, a) \tag{6}$$

这样构建起来的决策树就是CART决策树

2.2.3 连续值处理 - 二分法（bi-partition) 和 C4.5 决策树

假设某个属性a是一个连续值(也就是说a取值数量是无限的)。 简单的处理方法就是确定一个中点

$$a_{middle} = \frac{a_i + a_{i+1}}{2} 1 \leq i \leq n-1 \tag{7}$$

a的取值小于中点的是一颗子树，大于等于中点的是另一棵子树。

使用该思路对Gain稍作修改， 就能处理连续值属性了， 这样生成的决策树叫做C4.5决策树

2.2.4 剪枝

总的来说， 剪枝的目的是防止过拟合。 同时， 由于决策树的树状特性， 剪枝同时还能减少预测的计算量。 剪枝是一个使用测试集修正决策树的过程。 具体围绕着"某个节点是否应该展开"工作。

2.2.4.1 预剪枝

预剪枝基于如下一个设计： 在以属性a做划分的时候， 计算划分前和划分后的验证集精度， 如果划分后验证集进度不如划分前， 就不划分。

优点：

- 有枝没有展开， 泛化性能一般来说会提高
- 训练和预测是计算开销变小

缺点：

- 本质是基于贪心的剪枝， 有些分支，虽然当下不能提高验证集精准度， 他的子分支却可以显著提高验证集精准度。 这样的策略带来了欠拟合的风险

2.2.4.2 后剪枝

后剪枝要求先生成一棵完整的决策树， 从该决策树最深处的非页节点开始， 递归计算：

- 该节点展开时候， 本树的验证集精度
- 该节点收起， 本树的验证集精度

依据验证集精度决定是否要展开这个节点。

递归计算他的父节点。

优点：

- 后剪枝一般来说能比预剪枝保留更多分支， 一般而言， 后剪枝决策树的欠拟合风险很小， 泛化性能往往优于预剪枝决策树。

缺点：

- 后剪枝在决策生成之后才开始， 需要额外的计算量。

2.3 关键源代码

2.3.1 决策树构建

思路来自于2.2.1节伪代码

```
def dicision_tree_init(X, Y, attrs, is_con_array, root, purity_cal):
    # 递归基
    if len(set(Y)) == 1:
        root.attr = np.pi
        root.label = Y[0]
        return None

    # @todo:
    # 什么是D中样本在A上取值相同
    if len(attrs) == 0 or is_same_on_attr(X, attrs):
        root.attr = np.pi
        # Y 中出现次数最多的label设定为node的label
        root.label = np.argmax(np.bincount(Y))
        return None

    # 计算每个attr的划分收益
    purity_attrs = []
    for i, a in enumerate(attrs):
        is_con = is_con_array[i]
        p = purity_cal(X, Y, a, is_con)
        purity_attrs.append(p)

    chosen_index = purity_attrs.index(max(purity_attrs))
    chosen_attr = attrs[chosen_index]
    is_attr_conti = is_con_array[chosen_index]
    root.attr = chosen_attr
    root.label = np.pi
    print("chose", chosen_attr)
```

```
    del attrs[chosen_index]
    del is_con_array[chosen_index]

    x_attr_col = X[:, chosen_attr]
    # 离散数据处理
    if not is_attr_conti:
        for x_v in set(X[:, chosen_attr]):
            n = Node(-1, -1, x_v)
            root.children.append(n)
            # 不可能Dv empty 要是empty压根不会在set里
            # 选出 X[attr] == x_v的行

            index_x_equal_v = np.where(x_attr_col == x_v)
            X_x_equal_v = X[index_x_equal_v]
            Y_x_equal_v = Y[index_x_equal_v]
            dicision_tree_init(X_x_equal_v, Y_x_equal_v, attrs, is_con_array, n,
purity_cal)
    else:
        half = gain_cal_t(X, Y, chosen_attr)[0]
        n_l = Node(-1, -1, -np.inf)
        n_ge = Node(-1, -1, half)
        root.children.append(n_l)
        root.children.append(n_ge)

        index_x_less_half = np.where(x_attr_col < half)
        dicision_tree_init(X[index_x_less_half], Y[index_x_less_half], attrs, is_con_array,
n_l, purity_cal)

        index_x_ge_half = np.where(x_attr_col >= half)
        dicision_tree_init(X[index_x_ge_half], Y[index_x_ge_half], attrs, is_con_array,
n_ge, purity_cal)
```

2.3.2 Gain函数

信息熵计算

```
def ent(D):
    # D is a 1d np array which actually is Y
    s = 0
    for k in set(D):
        p_k = np.sum(np.where(D == k, 1, 0)) / np.shape(D)[0]
        if p_k == 0:
            # 此时Pklog2Pk 定义为 0
            continue
        s += p_k * np.log2(p_k)
    return -s
```

Gain函数---能够处理连续数值

```
def gain(X, Y, attr, is_conti):
    # X, Y 是numpy arrary attr是某个特征的index
    x_attr_col = X[:, attr]
```

```
    ent_Dv = []
    weight_Dv = []
    # 离散值处理 和公式一致， 不多赘述
    if not is_conti:
        for x_v in set(x_attr_col):
            index_x_equal_v = np.where(x_attr_col == x_v)
            y_x_equal_v = Y[index_x_equal_v]
            ent_Dv.append(ent(y_x_equal_v))
            weight_Dv.append(np.shape(y_x_equal_v)[0] / np.shape(Y)[0])
        return ent(Y) - np.sum(np.array(ent_Dv) * np.array(weight_Dv))
    # 连续值处理
    else:
        return gain_cal_t(X, Y, attr)[1]
```

gain_cal_t选择出最合适的t， 同时返回对应的gain值 — 针对连续值的处理方式

```
def gain_cal_t(X, Y, attr):
    x_attr_col = X[:, attr]
    G = []
    T = []
    for i in range(len(x_attr_col) - 1):
        t = (x_attr_col[i] + x_attr_col[i+1]) / 2
        ga = gain_continue(X, Y, attr, t)
        T.append(t)
        G.append(ga)
    best_t_index = np.argmax(G)
    return T[best_t_index], G[best_t_index]
```

2.3.3 Gini函数

我也实现了使用Gini函数做划分选择， 为了方便统一使用argmax， 我的Gini函数返回的是-Gini

```
def Gini(D):
    # return Gini
    s = []
    for k in set(D):
        p_k = np.sum(np.where(D == k, 1, 0)) / np.shape(D)[0]
        s.append(p_k)
    s = np.array(s)
    return 1 - np.sum(s * s)
```

gini_index 返回的是负数

```
def gini_index(X, Y, attr):
    # return -gini index to use argmax
    x_attr_col = X[:, attr]
    gini_Dv = []
    weight_Dv = []
    for x_v in set(x_attr_col):
        D_x_equal_v = np.where(x_attr_col == x_v)
        y_x_equal_v = Y[D_x_equal_v]
        gini_Dv.append(Gini(y_x_equal_v))
        weight_Dv.append(np.shape(y_x_equal_v)[0] / np.shape(Y)[0])

    return -np.sum(np.array(gini_Dv) * np.array(weight_Dv))
```

2.3.4 预剪枝

预剪枝算法和普通生成算法差别不大， 主要就集中在计算是不是要展开分支上， 在这里只放核心代码， 具体可以查看`hw006.py`的函数`dicision_tree_init_pre_pru`

```
# 计算不展开的验证集精确度
    root.label = cal_label(Y)
    root.attr = np.pi
    y_predict = []
    for i in range(X_test.shape[0]):
        x = X_test[i]
        y_p = dicision_tree_predict(x, real_root, [False, False, False, False, True])
        y_predict.append(y_p)

    acc_no = accuracy_score(Y_test, y_predict)

    # 计算展开的验证集收益
    root.attr = chosen_attr
    root.label = np.pi
    x_attr_col = X[:, chosen_attr]

    if not is_attr_conti:
        for x_v in set(X[:, chosen_attr]):
            n = Node(-1, -1, x_v)
            # 不可能Dv empty 要是empty压根不会在set里
            # 选出 X[attr] == x_v的行
            index_x_equal_v = np.where(x_attr_col == x_v)
            X_x_equal_v = X[index_x_equal_v]
            Y_x_equal_v = Y[index_x_equal_v]
            n.attr = np.pi
            n.label = cal_label(Y_x_equal_v)
            root.children.append(n)
    else:
        half = gain_cal_t(X, Y, chosen_attr)[0]
        n_l = Node(-1, -1, -np.inf)
        n_ge = Node(-1, -1, half)


        index_x_less_half = np.where(x_attr_col < half)
```

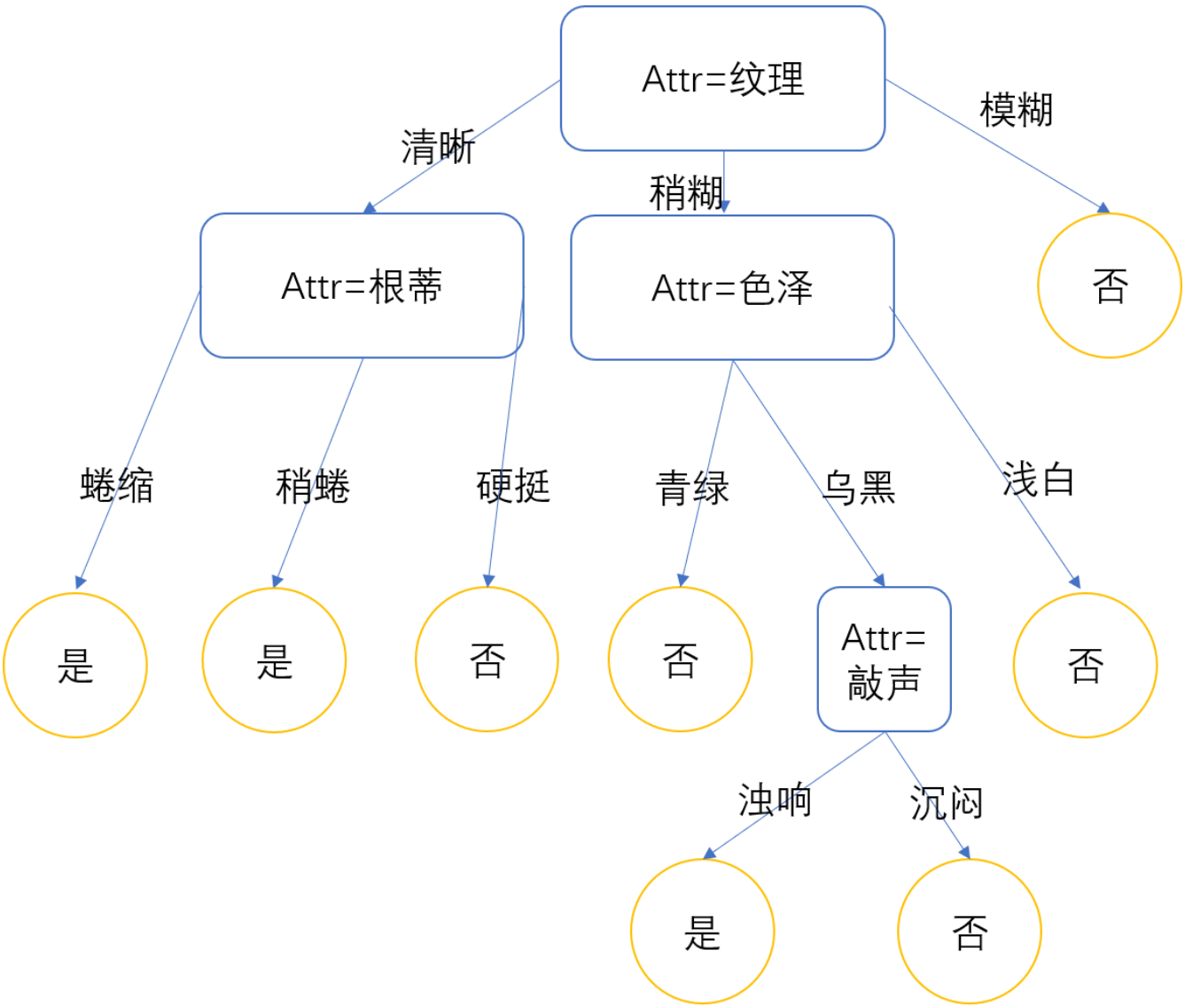
```
        n_l.attr = np.pi
        n_l.label = cal_label(Y[index_x_less_half])

        index_x_ge_half = np.where(x_attr_col >= half)
        n_ge.attr = np.pi
        n_ge.label = cal_label(Y[index_x_ge_half])

        root.children.append(n_l)
        root.children.append(n_ge)

    y_predict_yes = []
    for i in range(X_test.shape[0]):
        x = X_test[i]
        y_p = dicision_tree_predict(x, real_root, [False, False, False, False, True])
        y_predict_yes.append(y_p)

    acc_yes = accuracy_score(Y_test, y_predict_yes)

    print("acc of expand=", acc_yes)
    print("acc of not expand=", acc_no)
```

3 实验分析

实验分析的标准比较简单，就是测试集精准率。 其实就是在测试集上计算正确率。

3.1 离散数据处理-ID3决策树

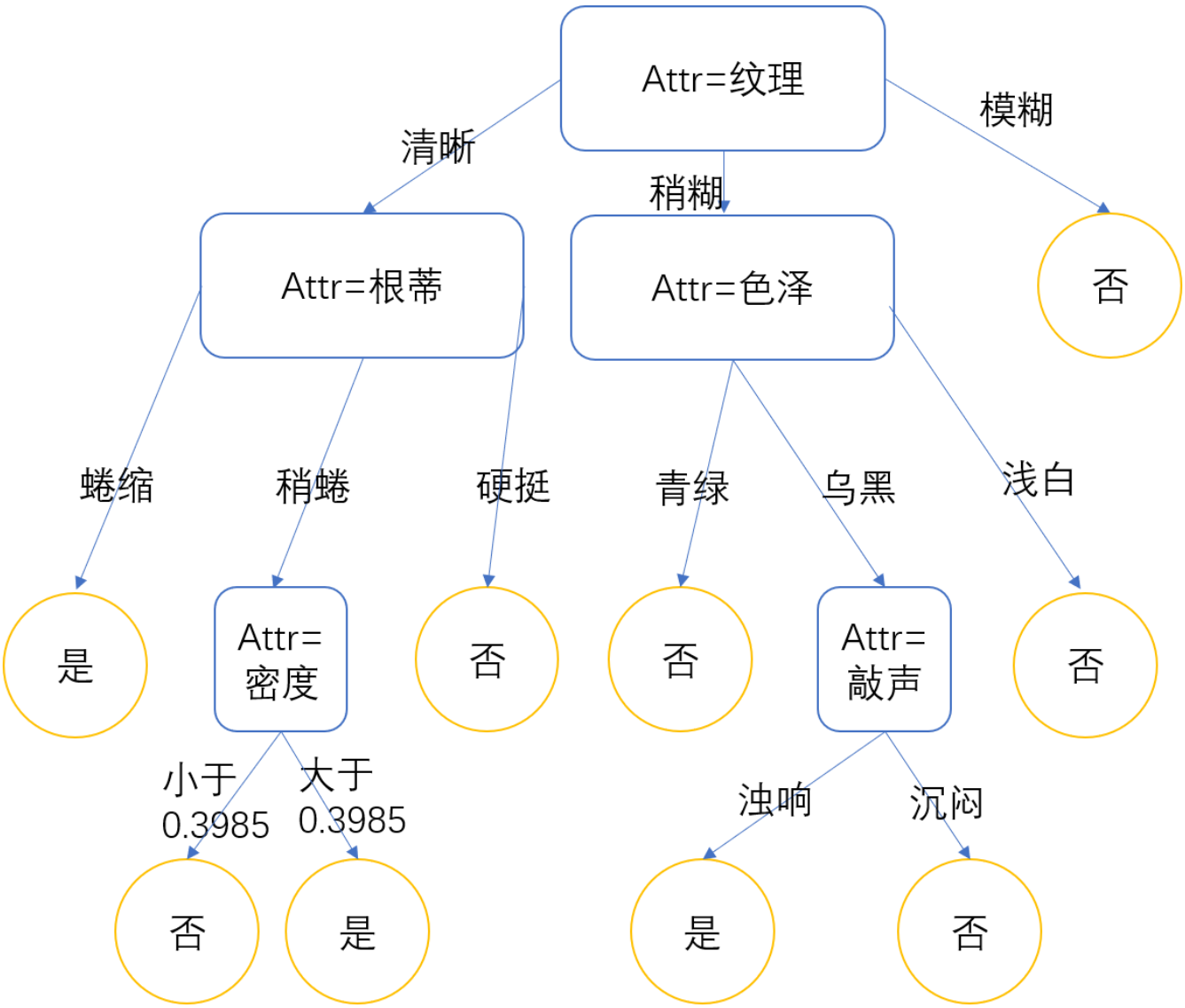
最后的测试集精准率为 0.75 根据生成过程画出决策树如下



使用Gini指数建立的树完全相同。

3.2 连续数据处理-C4.5决策树

最后的测试集精准率为0.6 决策树如下



3.3 预剪枝

收缩了决策树， 这有利于提高**模型的泛化能力**以及减少**预测计算时间。**

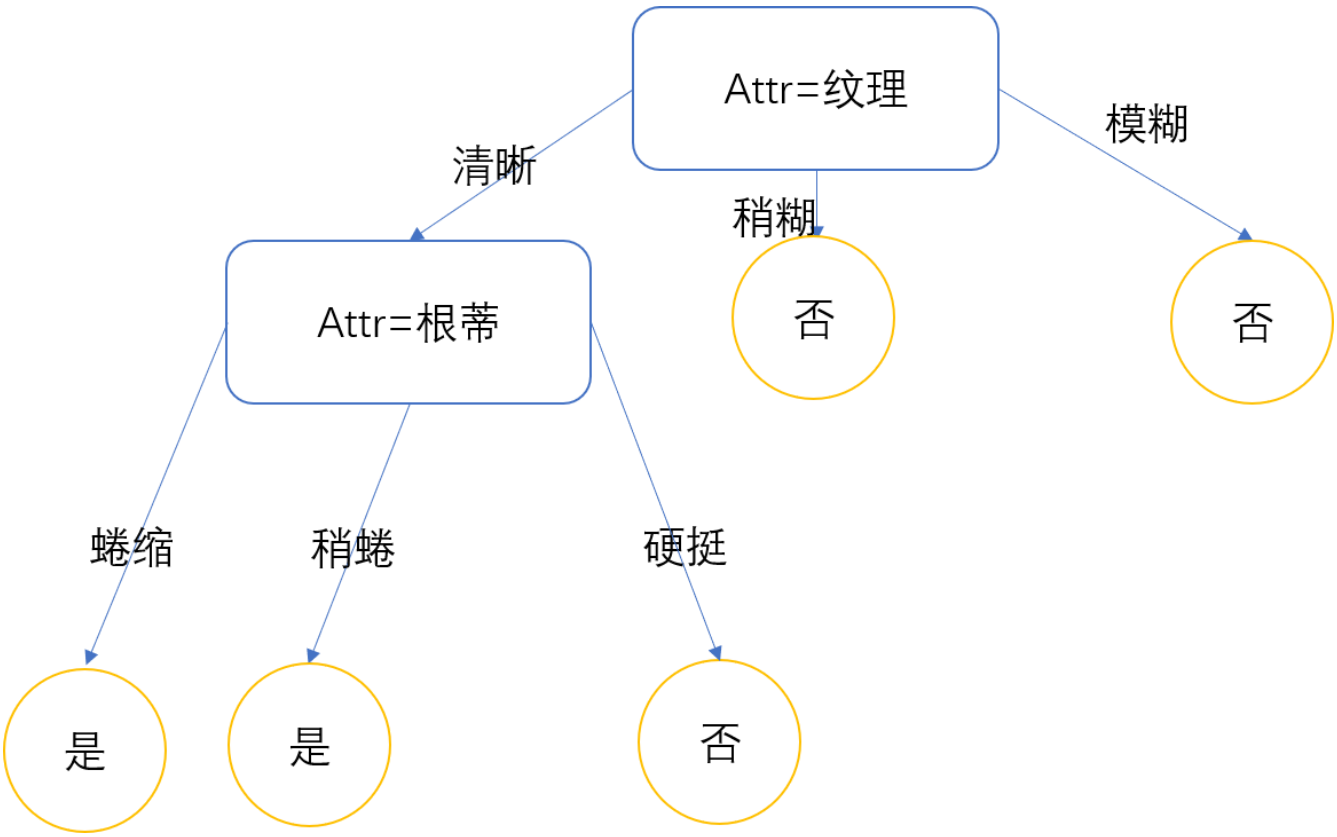
使用预剪枝有如下风险

- 预剪枝本身贪心的剪枝策略本身有不小的导致欠拟合的风险(2.2.4.1节中提到)
- 我们的数据集太小了， 训练集十几个sample，还要划分训练集和测试集拿来预剪枝， 进一步加大了欠拟合的风险。

3.3.1 ID3 预剪枝

不剪枝的acc=0.75

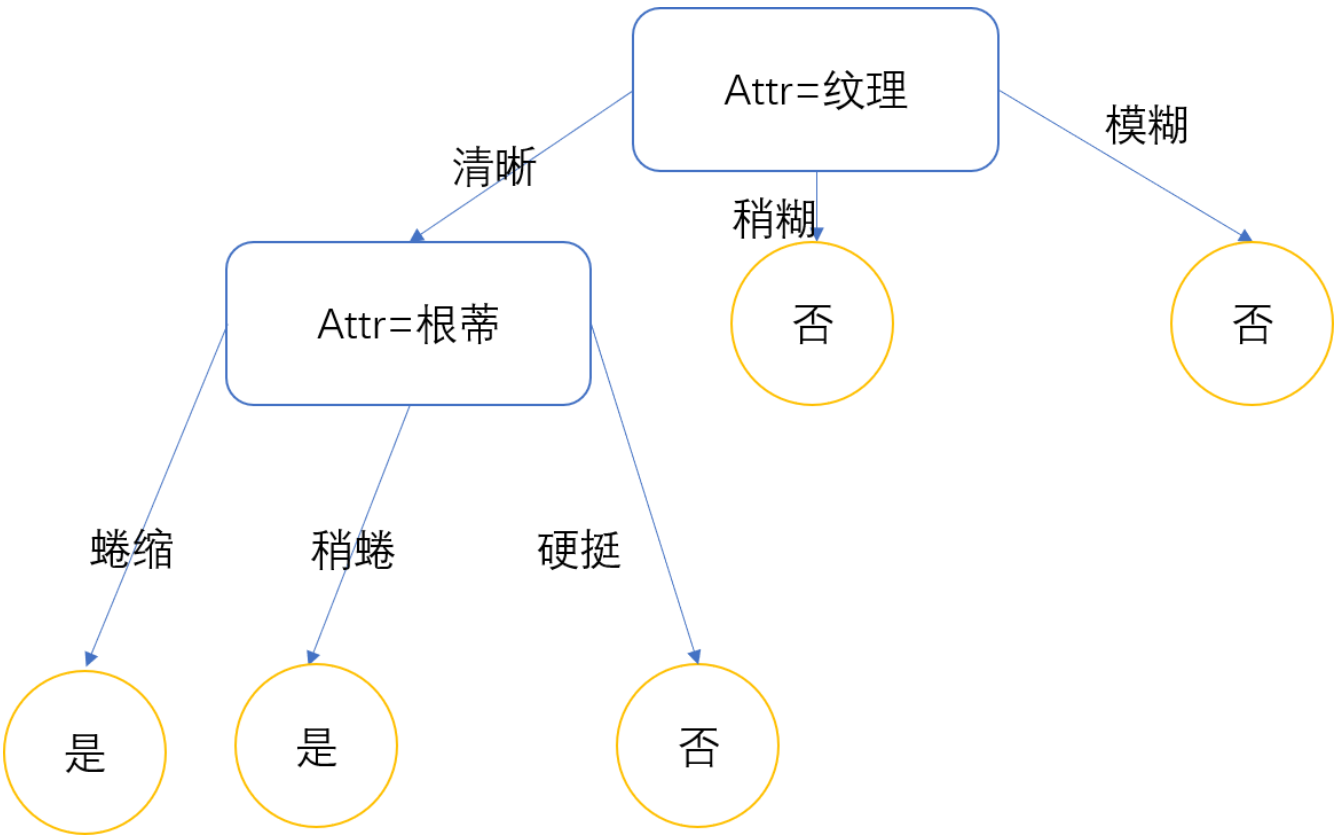
剪枝的acc=0.8



3.3.2 C4.5预剪枝

不剪枝的acc = 0.6

剪枝的acc = 0.8



不难看出， 两次预剪枝剪去的子分支有一样的部分：都剪去了纹理=稍糊的子分支。树二还减去了密度的子分支。

3.3.3 减去密度分支分析

我们抽出了训练集的编号1，3，13，15数据作为测试集

训练集

编号	色泽	根蒂	敲声	纹理	密度	好瓜
2	乌黑	蜷缩	沉闷	清晰	0.774	是
4	青绿	蜷缩	沉闷	清晰	0.608	是
5	浅白	蜷缩	浊响	清晰	0.556	是
6	青绿	稍蜷	浊响	清晰	0.403	是
7	乌黑	稍蜷	浊响	稍糊	0.481	是
8	乌黑	稍蜷	浊响	清晰	0.437	是
9	乌黑	稍蜷	沉闷	稍糊	0.666	否
10	青绿	硬挺	清脆	清晰	0.243	否
11	浅白	硬挺	清脆	模糊	0.245	否
12	浅白	蜷缩	浊响	模糊	0.343	否
14	浅白	稍蜷	沉闷	稍糊	0.657	否
16	浅白	蜷缩	浊响	模糊	0.593	否
17	青绿	蜷缩	沉闷	稍糊	0.719	否

测试集

编号	色泽	根蒂	敲声	纹理	密度	好瓜
1	青绿	蜷缩	浊响	清晰	0.697	是
3	乌黑	蜷缩	浊响	清晰	0.634	是
13	青绿	稍蜷	浊响	稍糊	0.639	否
15	乌黑	稍蜷	浊响	清晰	0.36	否

以纹理分， 纹理为清晰的sample有

[2，4，5，6，8，10]

再以根蒂分， 根蒂是稍蜷的sample有

[6, 8]

对这个sample计算最适合的分类attr=密度

计算分支前测试集精准度为0.75

计算分支后精准度 0.5

分支后测试集精准度反而变小。结论是不需要分支。剪枝。