机器学习第十次作业

```python
# 模块调用
import pandas as pd
import numpy as np
import treePlot
import Pruning

from sklearn.model_selection import train_test_split

from functools import reduce

from sklearn.utils.multiclass import type_of_target
```

```python
# 载入数据
melon_dataset = pd.read_excel('C:/Users/mi/Desktop/melon.xlsx')

# 划分数据集
X = melon_dataset.iloc[:,1:-1]
y = melon_dataset.iloc[:,-1]
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.4,stratify
= y,random_state = 1)
```

```
编号   色泽   根蒂   敲声   纹理   脐部   触感       密度       含糖率  好瓜
```

0 1 青绿 蜷缩 浊响 清晰 凹陷 硬滑 0.697 0.460 是
1 2 乌黑 蜷缩 沉闷 清晰 凹陷 硬滑 0.774 0.376 是
2 3 乌黑 蜷缩 浊响 清晰 凹陷 硬滑 0.634 0.264 是
3 4 青绿 蜷缩 沉闷 清晰 凹陷 硬滑 0.608 0.318 是
4 5 浅白 蜷缩 浊响 清晰 凹陷 硬滑 0.556 0.215 是
5 6 青绿 稍蜷 浊响 清晰 稍凹 软粘 0.403 0.237 是
6 7 乌黑 稍蜷 浊响 稍糊 稍凹 软粘 0.481 0.149 是
7 8 乌黑 稍蜷 浊响 清晰 稍凹 硬滑 0.437 0.211 是
8 9 乌黑 稍蜷 沉闷 稍糊 稍凹 硬滑 0.666 0.091 否
9 10 青绿 硬挺 清脆 清晰 平坦 软粘 0.243 0.267 否
10 11 浅白 硬挺 清脆 模糊 平坦 硬滑 0.245 0.057 否
11 12 浅白 蜷缩 浊响 模糊 平坦 软粘 0.343 0.099 否
12 13 青绿 稍蜷 浊响 稍糊 凹陷 硬滑 0.639 0.161 否
13 14 浅白 稍蜷 沉闷 稍糊 凹陷 硬滑 0.657 0.198 否
14 15 乌黑 稍蜷 浊响 清晰 稍凹 软粘 0.360 0.370 否
15 16 浅白 蜷缩 浊响 模糊 平坦 硬滑 0.593 0.042 否
16 17 青绿 蜷缩 沉闷 稍糊 稍凹 硬滑 0.719 0.103 否

DecisionTree.py

1. 定义结点类

```python
class Node(object):
    def __init__(self):
        # 属性名称
        self.feature_name = None
        # 属性编号(降低每个结点所占内存)
        self.feature_index = None
        # 子树集合 (dict: {featuretype: subtree})
        self.subtree = {}
        # 正例数占比
        self.impurity = None
        # 属性是否为连续变量
        self.is_continuous = False
        # 若为连续变量，定义临界值
        self.split_value = None
        # 是否为叶结点
        self.is_leaf = False
        # （作为叶结点）结点的类型
        self.leaf_class = None
        #  当前根节点对应决策树的叶子数
        self.leaf_num = None
        # 结点深度初始为-1
        self.high = -1
```

2. 定义决策树类

```python
# 决策树类
class Decision_Tree(object):
    # 不处理缺失值
    # 支持连续值情形
    # 采用信息增益作为划分依据
    def __init__(self, criterion = 'info_gain', pruning = None):
        #: param criterion: 划分方式选择，目前仅支持‘info_gain’信息增益
        #: param pruning: 是否剪枝，可选择‘pre_pruning’ 与 ‘post_pruning’
        # 检验参数合法性
        assert criterion in ('gini_index','info_gain','gain_ratio')
        assert pruning in (None, 'pre_pruning','post_pruning')
        self.criterion = criterion
        self.pruning = pruning

    def fit(self, X_train, y_train, X_valid = None, y_valid = None):
        #: param X_train: DataFrame类型数据 特征集合
        #: param y_train: DataFrame类型数据 分类标签
        #: param X_valid: DataFrame类型数据 剪枝特征集合
        #: param y_train: DataFrame类型数据 剪枝分类标签

        # 选择剪枝却未传入验证集
        if self.pruning is not None and (X_valid is None or y_valid is None):
            raise Exception('Please input validation data for pruning')

        # 输入验证集
        if X_valid is not None:
            pass

        # 存储特征名称
        self.columns = list(X_train.columns)

        # 建立决策树
```

```
        self.tree = self.generate_tree(X_train,y_train)

        # 预剪枝
        if self.pruning == 'pre_pruning':
            Pruning.pre_pruning(X_train, y_train, X_valid, y_valid, self.tree)

        # 后剪枝
        if self.pruning == 'post_pruning':
            Pruning.post_pruning(X_train, y_train, X_valid, y_valid, self.tree)

        return self

    def generate_tree(self, X, y):
        #: param X: DataFrame类型训练数据 特征集合
        #: param y: DataFrame类型训练数据 分类标签
        # 初始化根节点
        my_tree = Node()
        my_tree.leaf_num = 0

        ###################### 递归终止条件 ##################################
        # 样本全属于同一类别
        if y.nunique() == 1:
            # 将node标记为该类别的叶结点
            my_tree.is_leaf = True
            my_tree.leaf_class = y.values[0]
            # 根节点深度为0
            my_tree.high = 0
            # 根节点编号为1
            my_tree.leaf_num += 1
            return my_tree

        # 属性集为空或样本在属性上取值相同
        if X.empty or reduce(lambda x,y: x and y,(X.nunique().values ==
[1]*X.nunique().size)):
            # 标记为叶节点
            my_tree.is_leaf = True
            # 将样本中最多的类作为结点的类
            my_tree.leaf_class = pd.value_counts(y).index[0]
            my_tree.high = 0
            my_tree.leaf_num += 1
            return my_tree
        ######################################################################

        # 从属性集中选择最优划分属性和对应分化指标
        best_feature_name, best_impurity = self.choose_best_feature_to_split(X,
y)
        #print(best_feature_name)
        # 根节点命名
        my_tree.feature_name = best_feature_name
        my_tree.feature_impurity = best_impurity
        my_tree.feature_index = self.columns.index(best_feature_name)
        # 获得该属性的所有类别
        feature_values = X.loc[:, best_feature_name]

        # 特征离散,info_gain 函数返回一个长度为1的list
        if len(best_impurity) == 1:
            my_tree.is_continuous = False
            # 递归调用需要更新X
```

```python
            unique_vals = pd.unique(feature_values)
            # 叶结点的训练集(只对分类属性做训练集切分)
            sub_X = X.drop(best_feature_name, axis = 1)
            # 初次调用最大值为-1
            max_high = -1
            for value in unique_vals:
                # 通过索引关系建立根节点和叶结点的联系 [value: subtree]
                # 传入特征取值value的样例
                my_tree.subtree[value] = self.generate_tree(sub_X[feature_values
== value], y[feature_values == value])
                # 记录子树最大深度
                if my_tree.subtree[value].high > max_high:
                    max_high = my_tree.subtree[value].high
                my_tree.leaf_num += my_tree.subtree[value].leaf_num

            my_tree.high = max_high + 1

        elif len(best_impurity) == 2:
             my_tree.is_continuous = True
             my_tree.split_value = best_impurity[1]
             # 通过索引关系建立根节点和叶结点的联系 ['feature >= split_value':subtree]
             greater_part = '>= {:.3f}'.format(my_tree.split_value)
             less_part = '< {:.3f}'.format(my_tree.split_value)
             #print(my_tree.split_value)
             my_tree.subtree[greater_part] = self.generate_tree(X[feature_values
>= my_tree.split_value], y[feature_values >= my_tree.split_value])
             my_tree.subtree[less_part] = self.generate_tree(X[feature_values <
my_tree.split_value], y[feature_values < my_tree.split_value])

             # 连续问题的一次分类只会生成两棵子树
             my_tree.leaf_num = (my_tree.subtree[greater_part].leaf_num +
my_tree.subtree[less_part].leaf_num)
             my_tree.high = max(my_tree.subtree[greater_part].high,
my_tree.subtree[less_part].high) + 1

        return my_tree




    def choose_best_feature_to_split(self, X, y):
        # 检查划分依据合法性
        assert self.criterion in ('gini_index','info_gain','gain_ratio')

        # 根据基尼系数划分
        if self.criterion == 'gini_index':
            pass

        # 根据信息增益划分
        elif self.criterion == 'info_gain':
            return self.choose_best_feature_info_gain(X,y)

        # 根据增益比划分
        elif self.criterion == 'gain_ratio':
            pass


    def choose_best_feature_info_gain(self,X,y):
```

```
        #: param X: DataFrame类型训练数据 特征集合
        #: param y: DataFrame类型训练数据 分类标签
        #: return: [best_feature_name, best_info_gain]

        features = X.columns
        best_feature_name = None
        # 查找最大信息增益
        best_info_gain = [float('-inf')]
        # 计算样本的熵
        entD = self.entropy(y)
        # 计算各个属性的信息增益
        for feature_name in features:
            # 先判断是否为连续值
            # 返回值作为函数的参数而非结点属性
            is_continuous = type_of_target(X[feature_name]) == 'continuous'
            info_gain = self.info_gain(X[feature_name], y, entD, is_continuous)
            # 找到最大信息增益
            if info_gain[0] > best_info_gain[0]:
                best_feature_name = feature_name
                best_info_gain = info_gain

        return best_feature_name, best_info_gain


    def entropy(self,y):
        # 计算熵
        #: param y: 训练样本的分类标签
        # 计算各类的概率向量
        p_vector = pd.value_counts(y).values/y.shape[0]
        ent = np.sum(-p_vector * np.log2(p_vector))
        return ent


    def info_gain(self, X_feature, y, entD, is_continuous = False):
        # 计算信息增益
        #: param X_feature : DataFrame类型训练数据 某一特征集合
        #: param y: DataFrame类型训练数据 分类标签
        #: param entD: 结点信息熵
        #: param is_continuous: 特征类型
        #: return 1.连续变量 [gain, min_ent_point]
        #         2.分类变量 [gain]
        unique_value = pd.unique(X_feature)
        if is_continuous:
            # 变量为连续类型
            # 避免出现相同分界值
            unique_value.sort()
            split_point_set = [(unique_value[i] + unique_value[i + 1])/2 for i
in range(len(unique_value) - 1)]
            # 最小条件熵
            min_ent = float('inf')
            # 最小条件熵对应分界点
            min_ent_point = None
            for split_point in split_point_set:
                Dv1 = y[X_feature <= split_point]
                Dv2 = y[X_feature > split_point]
                feature_ent = Dv1.shape[0] / y.shape[0] * self.entropy(Dv1) +
Dv2.shape[0] / y.shape[0] * self.entropy(Dv2)
```

```python
                # 找到最小条件熵
                if feature_ent < min_ent:
                    min_ent = feature_ent
                    min_ent_point = split_point

            gain = entD - min_ent
            return [gain, min_ent_point]

        else:
            feature_ent = 0
            # 直接计算条件熵
            for value in unique_value:
                Dv = y[X_feature == value]
                feature_ent += Dv.shape[0] / y.shape[0] * self.entropy(Dv)

            gain = entD - feature_ent
            return [gain]

    def predict(self, X):
        #: param X : DataFrame类型测试数据
        #: return 若测试数据只有1条，返回值
        #          若有多条，返回向量
        # 检查实例中是否存在tree属性(是否已经拟合训练集数据)
        if not hasattr(self, "tree"):
            raise Exception('Please fit the data to generate a tree')

        if X.ndim == 1:
            return self.predict_single(X)
        else:
            return X.apply(self.predict_single, axis = 1)

    def predict_single(self, x, subtree = None):
        # 预测单一样例
        #:param x: 单一样例
        #:subtree 子树(预测起点的根节点)
        #:return

        # 默认从整棵树的根节点找起
        if subtree is None:
            subtree = self.tree

        # 子树为叶结点，返回叶结点类型作为预测结果
        if subtree.is_leaf:
            return subtree.leaf_class

        # 子树属性为连续变量
        if subtree.is_continuous:  # 若是连续值，需要判断是

            if x[subtree.feature_index] >= subtree.split_value:
                # 子树有字典类型属性 subtree
                return self.predict_single(x, subtree.subtree['>=
{:.3f}'.format(subtree.split_value)])
            else:
                return self.predict_single(x, subtree.subtree['<
{:.3f}'.format(subtree.split_value)])
        else:
```

```
            return self.predict_single(x,
subtree.subtree[x[subtree.feature_index]])
```

Pruning.py

```
import pandas as pd
import numpy as np


def post_pruning(X_train, y_train, X_val, y_val, tree = None):

        # 若剪枝对象是叶结点
        if tree.is_leaf:
            return tree

        # 若验证集为空集，则不再进行剪树枝
        if X_val.empty:
            return tree

        # 找到分支结点样例中含最多样例的类别标签
        most_common_in_train = pd.value_counts(y_train).index[0]
        # 计算当前的分类精度
        current_accuracy = np.mean(y_val == most_common_in_train)

        if tree.is_continuous:
            # 剪枝属性连续
            # 找出当前属性对应叶结点中的样例
            greater_part_train = X_train.loc[:, tree.feature_name] >=
tree.split_value
            less_part_train = X_train.loc[:, tree.feature_name] <
tree.split_value

            greater_part_val = X_val.loc[:, tree.feature_name] >=
tree.split_value
            less_part_val = X_val.loc[:, tree.feature_name] < tree.split_value

            # greater_subtree指向当前连续分支结点的左结点
            greater_subtree = post_pruning(X_train[greater_part_train],
y_train[greater_part_train], X_val[greater_part_val], y_val[greater_part_val],
tree.subtree['>= {:.3f}'.format(tree.split_value)])
            tree.subtree['>= {:.3f}'.format(tree.split_value)] = greater_subtree

            # less_subtree指向当前连续分支结点的右结点
            less_subtree = post_pruning(X_train[less_part_train],
y_train[less_part_train], X_val[less_part_val], y_val[less_part_val],
tree.subtree['< {:.3f}'.format(tree.split_value)])
            tree.subtree['< {:.3f}'.format(tree.split_value)] = less_subtree

            # 记录树的高度
            tree.high = max(greater_subtree.high, less_subtree.high) + 1
            # 记录树的叶结点个数
            tree.leaf_num = (greater_subtree.leaf_num + less_subtree.leaf_num)

            # tree指向最深分支结点，子节点均为叶结点
            if greater_subtree.is_leaf and less_subtree.is_leaf:
                # 定义分划函数
```

```
                def split_fun(x):
                    if x >= tree.split_value:
                        return '>= {:.3f}'.format(tree.split_value)
                    else:
                        return '< {:.3f}'.format(tree.split_value)

                # 给出每一个样例的划分结果
                val_split = X_val.loc[:, tree.feature_name].map(split_fun)
                # 判断叶结点中样例是否分类正确，返回bool 向量
                right_class_in_val = y_val.groupby(val_split).apply(lambda x:
np.sum(x == tree.subtree[x.name].leaf_class))
                # 计算正确率
                split_accuracy = right_class_in_val.sum() / y_val.shape[0]

                # 若当前节点为叶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:
                # 将当前节点设为叶节点
                    set_leaf(pd.value_counts(y_train).index[0], tree)
        else:
         # 剪枝属性离散
            max_high = -1
            tree.leaf_num = 0
            # 判断当前节点下，所有子树是否都为叶节点
            is_all_leaf = True

            for key in tree.subtree.keys():
                    # 遍历所有子树
                    # 找到对应子树数据集的bool索引
                    this_part_train = X_train.loc[:, tree.feature_name] == key
                    this_part_val = X_val.loc[:, tree.feature_name] == key

                    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])

                    if tree.subtree[key].high > max_high:
                        max_high = tree.subtree[key].high
                    tree.leaf_num += tree.subtree[key].leaf_num

                    if not tree.subtree[key].is_leaf:
                    # 若有一个子树不是叶节点
                        is_all_leaf = False

                    tree.high = max_high + 1

            if is_all_leaf:
                    # 若所有子节点都为叶节点，则考虑是否进行剪枝
                    # 判断叶结点中样例是否分类正确，返回bool 向量
                    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))
                    # 计算正确率
                    split_accuracy = right_class_in_val.sum() / y_val.shape[0]

                    if current_accuracy > split_accuracy:
                        # 若当前节点为叶节点时的准确率大于不剪枝的准确率，则进行剪枝操作——
将当前节点设为叶节点
                        set_leaf(pd.value_counts(y_train).index[0], tree)
```

```
        return tree



def pre_pruning(X_train, y_train, X_val, y_val, tree_=None):
# 预剪枝
    if tree_.is_leaf:
        # 若当前节点已经为叶节点，那么就直接return了
        return tree_


    if X_val.empty:
        # 验证集为空集时，不再剪枝
        return tree_

    # 在计算准确率时，由于西瓜数据集的原因，好瓜和坏瓜的数量会一样，这个时候选择训练集中样本最
多的类别时会不稳定（因为都是50%），
    # 导致准确率不稳定，当然在数量大的时候这种情况很少会发生。
    most_common_in_train = pd.value_counts(y_train).index[0]
    current_accuracy = np.mean(y_val == most_common_in_train)

    if tree_.is_continuous:  # 连续值时，需要将样本分割为两部分，来计算分割后的正确率
        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)

        if current_accuracy >= split_accuracy:
            # 当前节点为叶节点时准确率大于或分割后的准确率时，选择不划分
            set_leaf(pd.value_counts(y_train).index[0], tree_)

        else:
            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
            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
            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
            tree_.high = max(up_subtree.high, down_subtree.high) + 1
            tree_.leaf_num = (up_subtree.leaf_num + down_subtree.leaf_num)



    else:
        # 若是离散值，则变量所有值，计算分割后正确率
```

```python
        split_accuracy = val_accuracy_after_split(X_train[tree_.feature_name],
y_train,X_val[tree_.feature_name], y_val)

        if current_accuracy >= split_accuracy:
            set_leaf(pd.value_counts(y_train).index[0], tree_)
        else:
            max_high = -1
            tree_.leaf_num = 0
            for key in tree_.subtree.keys():
                this_part_train = X_train.loc[:, tree_.feature_name] == key
                this_part_val = X_val.loc[:, tree_.feature_name] == key
                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])
                if tree_.subtree[key].high > max_high:
                    max_high = tree_.subtree[key].high
                tree_.leaf_num += tree_.subtree[key].leaf_num
            tree_.high = max_high + 1
    return tree_



def set_leaf(leaf_class, tree_):
    # 设置节点为叶节点
    tree_.is_leaf = True
    # 若划分前正确率大于划分后正确率。则选择不划分，将当前节点设置为叶节点
    tree_.leaf_class = leaf_class
    tree_.feature_name = None
    tree_.feature_index = None
    tree_.subtree = {}
    tree_.impurity = None
    tree_.split_value = None
    tree_.high = 0  # 重新设立高 和叶节点数量
    tree_.leaf_num = 1


def val_accuracy_after_split(feature_train, y_train, feature_val, y_val,
split_value=None):
    # 若是连续值时，需要需要按切分点对feature 进行分组，若是离散值，则不用处理
    if split_value is not None:
        def split_fun(x):
            if x >= split_value:
                return '>= {:.3f}'.format(split_value)
            else:
                return '< {:.3f}'.format(split_value)

        train_split = feature_train.map(split_fun)
        val_split = feature_val.map(split_fun)
    else:
        train_split = feature_train
        val_split = feature_val

    majority_class_in_train = y_train.groupby(train_split).apply(lambda x:
pd.value_counts(x).index[0])
    # 计算各特征下样本最多的类别
    right_class_in_val = y_val.groupby(val_split).apply(lambda x: np.sum(x ==
majority_class_in_train[x.name]))  # 计算各类别对应的数量
    # 返回准确率
```

```
    return right_class_in_val.sum() / y_val.shape[0]
```

treePlot.py

```
from matplotlib import pyplot as plt

# 设置绘图字体
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

decision_node = dict(boxstyle='round,pad=0.3', fc='#FAEBD7')
leaf_node = dict(boxstyle='round,pad=0.3', fc='#F4A460')
arrow_args = dict(arrowstyle="<-")

y_off = None
x_off = None
total_num_leaf = None
total_high = None


def plot_node(node_text, center_pt, parent_pt, node_type, ax_):
    ax_.annotate(node_text, xy=[parent_pt[0], parent_pt[1] - 0.02],
xycoords='axes fraction',
                 xytext=center_pt, textcoords='axes fraction',
                 va="center", ha="center", size=15,
                 bbox=node_type, arrowprops=arrow_args)


def plot_mid_text(mid_text, center_pt, parent_pt, ax_):
    x_mid = (parent_pt[0] - center_pt[0]) / 2 + center_pt[0]
    y_mid = (parent_pt[1] - center_pt[1]) / 2 + center_pt[1]
    ax_.text(x_mid, y_mid, mid_text, fontdict=dict(size=10))


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
```

```python
                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


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()
    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()
```

main.py

```python
# 训练集
print('训练集：')
print(X_train)

# 验证集
print('验证集：')
print(X_test)
```

- 4.2数据集生成的决策树在剪枝后形状不太理想，这里采用了自定义的训练集和验证集

训练集：

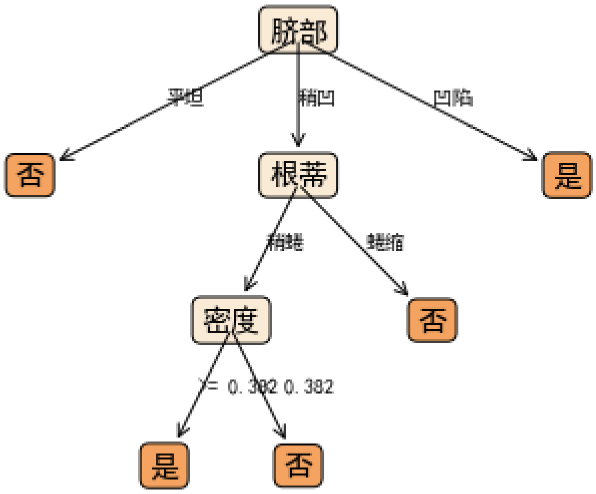
	色泽	根蒂	敲声	纹理	脐部	触感	密度	含糖率
10	浅白	硬挺	清脆	模糊	平坦	硬滑	0.245	0.057
5	青绿	稍蜷	浊响	清晰	稍凹	软粘	0.403	0.237
15	浅白	蜷缩	浊响	模糊	平坦	硬滑	0.593	0.042
16	青绿	蜷缩	沉闷	稍糊	稍凹	硬滑	0.719	0.103
3	青绿	蜷缩	沉闷	清晰	凹陷	硬滑	0.608	0.318
2	乌黑	蜷缩	浊响	清晰	凹陷	硬滑	0.634	0.264
4	浅白	蜷缩	浊响	清晰	凹陷	硬滑	0.556	0.215
9	青绿	硬挺	清脆	清晰	平坦	软粘	0.243	0.267
14	乌黑	稍蜷	浊响	清晰	稍凹	软粘	0.360	0.370
6	乌黑	稍蜷	浊响	稍糊	稍凹	软粘	0.481	0.149

验证集：

	色泽	根蒂	敲声	纹理	脐部	触感	密度	含糖率
13	浅白	稍蜷	沉闷	稍糊	凹陷	硬滑	0.657	0.198
0	青绿	蜷缩	浊响	清晰	凹陷	硬滑	0.697	0.460
12	青绿	稍蜷	浊响	稍糊	凹陷	硬滑	0.639	0.161
8	乌黑	稍蜷	沉闷	稍糊	稍凹	硬滑	0.666	0.091
11	浅白	蜷缩	浊响	模糊	平坦	软粘	0.343	0.099
7	乌黑	稍蜷	浊响	清晰	稍凹	硬滑	0.437	0.211
1	乌黑	蜷缩	沉闷	清晰	凹陷	硬滑	0.774	0.376

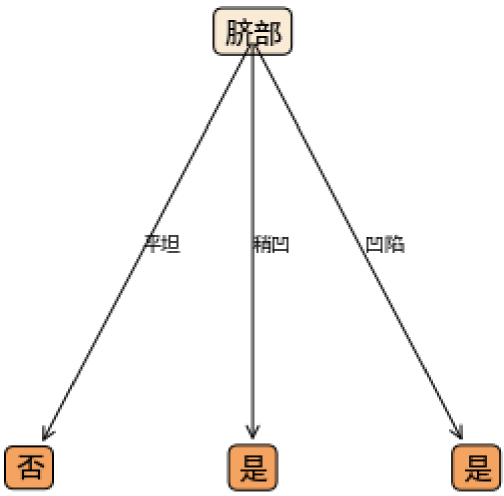
```python
#  不剪枝
tree1 = Decision_Tree()
tree1.fit(X_train, y_train, X_test, y_test)

treePlot.create_plot(tree1.tree)
```



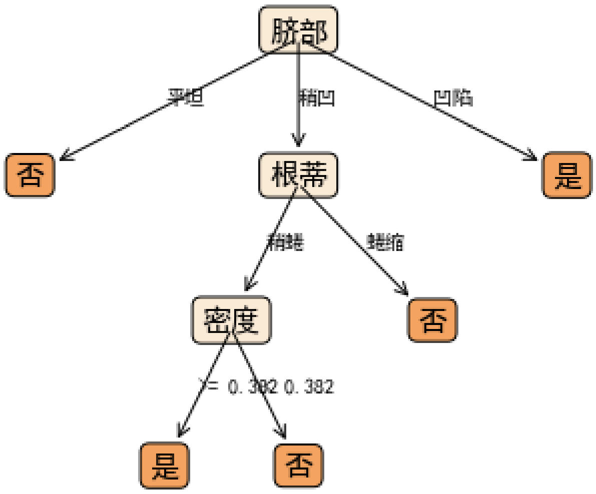

```python
#预剪枝
tree2 = Decision_Tree(pruning = 'pre_pruning')
tree2.fit(X_train, y_train, X_test, y_test)

treePlot.create_plot(tree2.tree)
```




```python
#后剪枝
tree3 = Decision_Tree(pruning = 'post_pruning')
tree3.fit(X_train, y_train, X_test, y_test)

treePlot.create_plot(tree3.tree)
```


脐部
平坦
稍凹
凹陷
否
根蒂
是
稍蜷
蜷缩
密度
否
>= 0.382
0.382
是
否