机器学习第四章课后习题

4.1

决策树中树枝停止生长生成叶节点，只会发生在样本属于同一类或是所有特征值都用完的情况，那么可能导致叶节点标记与实际训练集不同时只会发生特征值都用完的情况，即同一节点中的样本，其路径上的特征值都是完全相同的，而由由于训练集中没有冲突数据，因此每个节点上的训练误差都为0，因此必定存在与训练集一致的决策树。

4.2

缺陷：使用"最小训练误差"作为划分选择准则的话，由于训练集总是和真实情况存在一定的偏差，这样会使得到的决策树存在过拟合的情况，对于未知的数据的泛化能力较差。

4.3-4.4

```
# treeplottter
from matplotlib import pyplot as plt

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
            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()
```

```
# 生成决策树
import numpy as np
import pandas as pd
from sklearn.utils.multiclass import type_of_target
import treePlottter
import pruning


class Node(object):
    def __init__(self):
        self.feature_name = None
        self.feature_index = None
        self.subtree = {}
```

```
        self.impurity = None
        self.is_continuous = False
        self.split_value = None
        self.is_leaf = False
        self.leaf_class = None
        self.leaf_num = None
        self.high = -1


class DecisionTree(object):
    '''
    没有针对缺失值的情况作处理。
    '''

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

        :param criterion: 划分方法选择，'gini', 'infogain', 'gainratio', 三种选项。
        :param pruning:   是否剪枝。 'pre_pruning' 'post_pruning'
        '''
        assert criterion in ('gini', 'infogain', 'gainratio')
        assert pruning in (None, 'pre_pruning', 'post_pruning')
        self.criterion = criterion
        self.pruning = pruning

    def fit(self, X_train, y_train, X_val=None, y_val=None):
        '''
        生成决策树
        -------
        :param X:  只支持DataFrame类型数据，因为DataFrame中已有列名，省去一个列名的参数。
不支持np.array等其他数据类型
        :param y:
        :return:
        '''

        if self.pruning is not None and (X_val is None or y_val is None):
            raise Exception('you must input X_val and y_val if you are going to
pruning')

        X_train.reset_index(inplace=True, drop=True)
        y_train.reset_index(inplace=True, drop=True)

        if X_val is not None:
            X_val.reset_index(inplace=True, drop=True)
            y_val.reset_index(inplace=True, drop=True)

        self.columns = list(X_train.columns)  # 包括原数据的列名
        self.tree_ = self.generate_tree(X_train, y_train)

        if self.pruning == 'pre_pruning':
            pruning.pre_pruning(X_train, y_train, X_val, y_val, self.tree_)
        elif self.pruning == 'post_pruning':
            pruning.post_pruning(X_train, y_train, X_val, y_val, self.tree_)

        return self
```

```
    def generate_tree(self, X, y):
        my_tree = Node()
        my_tree.leaf_num = 0
        if y.nunique() == 1:  # 属于同一类别
            my_tree.is_leaf = True
            my_tree.leaf_class = y.values[0]
            my_tree.high = 0
            my_tree.leaf_num += 1
            return my_tree

        if X.empty:  # 特征用完了，数据为空，返回样本数最多的类
            my_tree.is_leaf = True
            my_tree.leaf_class = pd.value_counts(y).index[0]
            my_tree.high = 0
            my_tree.leaf_num += 1
            return my_tree

        best_feature_name, best_impurity = self.choose_best_feature_to_split(X,
y)

        my_tree.feature_name = best_feature_name
        my_tree.impurity = best_impurity[0]
        my_tree.feature_index = self.columns.index(best_feature_name)

        feature_values = X.loc[:, best_feature_name]

        if len(best_impurity) == 1:  # 离散值
            my_tree.is_continuous = False

            unique_vals = pd.unique(feature_values)
            sub_X = X.drop(best_feature_name, axis=1)

            max_high = -1
            for value in unique_vals:
                my_tree.subtree[value] = self.generate_tree(sub_X[feature_values
== value], y[feature_values == value])
                if my_tree.subtree[value].high > max_high:  # 记录子树下最高的高度
                    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:  # 连续值
            my_tree.is_continuous = True
            my_tree.split_value = best_impurity[1]
            up_part = '>= {:.3f}'.format(my_tree.split_value)
            down_part = '< {:.3f}'.format(my_tree.split_value)

            my_tree.subtree[up_part] = self.generate_tree(X[feature_values >=
my_tree.split_value],
                                                          y[feature_values >=
my_tree.split_value])
            my_tree.subtree[down_part] = self.generate_tree(X[feature_values <
my_tree.split_value],
                                                            y[feature_values <
my_tree.split_value])
```

```
            my_tree.leaf_num += (my_tree.subtree[up_part].leaf_num +
my_tree.subtree[down_part].leaf_num)

            my_tree.high = max(my_tree.subtree[up_part].high,
my_tree.subtree[down_part].high) + 1

        return my_tree

    def predict(self, X):
        '''
        同样只支持 pd.DataFrame类型数据
        :param X:  pd.DataFrame 类型
        :return:   若
        '''
        if not hasattr(self, "tree_"):
            raise Exception('you have to fit first before predict.')
        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):
        '''
        预测单一样本。 实际上这里也可以写成循环，写成递归样本大的时候有栈溢出的风险。
        :param x:
        :param subtree: 根据特征，往下递进的子树。
        :return:
        '''
        if subtree is None:
            subtree = self.tree_

        if subtree.is_leaf:
            return subtree.leaf_class

        if subtree.is_continuous:  # 若是连续值，需要判断是
            if x[subtree.feature_index] >= subtree.split_value:
                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]])

    def choose_best_feature_to_split(self, X, y):
        assert self.criterion in ('gini', 'infogain', 'gainratio')

        if self.criterion == 'gini':
            return self.choose_best_feature_gini(X, y)
        elif self.criterion == 'infogain':
            return self.choose_best_feature_infogain(X, y)
        elif self.criterion == 'gainratio':
            return self.choose_best_feature_gainratio(X, y)
```

```
    def choose_best_feature_gini(self, X, y):
        features = X.columns
        best_feature_name = None
        best_gini = [float('inf')]
        for feature_name in features:
            is_continuous = type_of_target(X[feature_name]) == 'continuous'
            gini_idex = self.gini_index(X[feature_name], y, is_continuous)
            if gini_idex[0] < best_gini[0]:
                best_feature_name = feature_name
                best_gini = gini_idex

        return best_feature_name, best_gini

    def choose_best_feature_infogain(self, X, y):
        '''
        以返回值中best_info_gain 的长度来判断当前特征是否为连续值，若长度为 1 则为离散值，若长度为 2 ， 则为连续值
        :param X: 当前所有特征的数据 pd.DaraFrame格式
        :param y: 标签值
        :return:  以信息增益来选择的最佳划分属性，第一个返回值为属性名称，

        '''
        features = X.columns
        best_feature_name = None
        best_info_gain = [float('-inf')]
        entD = self.entroy(y)
        for feature_name in features:
            is_continuous = type_of_target(X[feature_name]) == 'continuous'
            info_gain = self.info_gain(X[feature_name], y, entD, is_continuous)
            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 choose_best_feature_gainratio(self, X, y):
        '''
        以返回值中best_gain_ratio 的长度来判断当前特征是否为连续值，若长度为 1 则为离散值，若长度为 2 ， 则为连续值
        :param X: 当前所有特征的数据 pd.DaraFrame格式
        :param y: 标签值
        :return:  以信息增益率来选择的最佳划分属性，第一个返回值为属性名称，第二个为最佳划分属性对应的信息增益率
        '''
        features = X.columns
        best_feature_name = None
        best_gain_ratio = [float('-inf')]
        entD = self.entroy(y)

        for feature_name in features:
            is_continuous = type_of_target(X[feature_name]) == 'continuous'
            info_gain_ratio = self.info_gainRatio(X[feature_name], y, entD,
is_continuous)
            if info_gain_ratio[0] > best_gain_ratio[0]:
                best_feature_name = feature_name
                best_gain_ratio = info_gain_ratio
```

```
        return best_feature_name, best_gain_ratio

    def gini_index(self, feature, y, is_continuous=False):
        '''
        计算基尼指数， 对于连续值，选择基尼系统最小的点，作为分割点
        -------
        :param feature:
        :param y:
        :return:
        '''
        m = y.shape[0]
        unique_value = pd.unique(feature)
        if is_continuous:
            unique_value.sort()  # 排序，用于建立分割点
            # 这里其实也可以直接用feature值作为分割点，但这样会出现空集， 所以还是按照书中4.7式建立分割点。好处是不会出现空集
            split_point_set = [(unique_value[i] + unique_value[i + 1]) / 2 for i in range(len(unique_value) - 1)]

            min_gini = float('inf')
            min_gini_point = None
            for split_point_ in split_point_set:  # 遍历所有的分割点，寻找基尼指数最小的分割点
                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]
        else:
            gini_index = 0
            for value in unique_value:
                Dv = y[feature == value]
                m_dv = Dv.shape[0]
                gini = self.gini(Dv)  # 原书4.5式
                gini_index += m_dv / m * gini  # 4.6式

            return [gini_index]

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

    def info_gain(self, feature, y, entD, is_continuous=False):
        '''
        计算信息增益
        ------
        :param feature: 当前特征下所有样本值
        :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]
        unique_value = pd.unique(feature)
        if is_continuous:
            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)]
            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_]
                Dv2 = y[feature > split_point_]
                feature_ent_ = Dv1.shape[0] / m * self.entroy(Dv1) + Dv2.shape[0]
/ m * self.entroy(Dv2)

                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
            for value in unique_value:
                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  # 原书中4.2式
            return [gain]

    def info_gainRatio(self, feature, y, entD, is_continuous=False):
        '''
        计算信息增益率 参数和info_gain方法中参数一致
        ------
        :param feature:
        :param y:
        :param entD:
        :return:
        '''

        if is_continuous:
            # 对于连续值，以最大化信息增益选择划分点之后，计算信息增益率，注意，在选择划分点之
后，需要对信息增益进行修正，要减去log_2(N-1)/|D|，N是当前特征的取值个数，D是总数据量。
            # 修正原因是因为：当离散属性和连续属性并存时，C4.5算法倾向于选择连续特征做最佳树分
裂点
            # 信息增益修正中，N的值，网上有些资料认为是“可能分裂点的个数”，也有的是“当前特征的
取值个数”，这里采用“当前特征的取值个数”。
            # 这样 (N-1)的值，就是去重后的“分裂点的个数” ，即在info_gain函数中，
split_point_set的长度，个人感觉这样更加合理。有时间再看看原论文吧。

            gain, split_point = self.info_gain(feature, y, entD, is_continuous)
            p1 = np.sum(feature <= split_point) / feature.shape[0]  # 小于或划分点
的样本占比
```

```
            p2 = 1 - p1  # 大于划分点样本占比
            IV = -(p1 * np.log2(p1) + p2 * np.log2(p2))

            grain_ratio = (gain - np.log2(feature.nunique()) / len(y)) / IV  # 对信息增益修正
            return [grain_ratio, split_point]
        else:
            p = pd.value_counts(feature) / feature.shape[0]  # 当前特征下 各取值样本所占比率
            IV = np.sum(-p * np.log2(p))  # 原书4.4式
            grain_ratio = self.info_gain(feature, y, entD, is_continuous)[0] / IV
            return [grain_ratio]

    def entroy(self, y):
        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


if __name__ == '__main__':
    # 4.4
    data_path2 = r'C:\Users\hanmi\Documents\xiguabook\watermelon2_0_Ch.txt'
    data = pd.read_table(data_path2, encoding='utf8', delimiter=',', index_col=0)

    train = [1, 2, 3, 6, 7, 10, 14, 15, 16, 17]
    train = [i - 1 for i in train]
    X = data.iloc[train, :6]
    y = data.iloc[train, 6]

    test = [4, 5, 8, 9, 11, 12, 13]
    test = [i - 1 for i in test]

    X_val = data.iloc[test, :6]
    y_val = data.iloc[test, 6]

    tree = DecisionTree('gini', 'pre_pruning')
    tree.fit(X, y, X_val, y_val)

    print(np.mean(tree.predict(X_val) == y_val))
    treePlottter.create_plot(tree.tree_)
```

```
# 剪枝
import pandas as pd
import numpy as np


def post_pruning(X_train, y_train, X_val, y_val, tree_=None):
    if tree_.is_leaf:
        return tree_

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

    most_common_in_train = pd.value_counts(y_train).index[0]
```

```
    current_accuracy = np.mean(y_val == most_common_in_train)  # 当前节点下验证集样本准确率

    if tree_.is_continuous:
        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 = 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
        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

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

        if up_subtree.is_leaf and down_subtree.is_leaf:
            def split_fun(x):
                if x >= tree_.split_value:
                    return '>= {:.3f}'.format(tree_.split_value)
                else:
                    return '< {:.3f}'.format(tree_.split_value)

            val_split = X_val.loc[:, tree_.feature_name].map(split_fun)
            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]

            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_)
    else:
        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

        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] = 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:
                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
            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]

            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):
    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

        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

    return right_class_in_val.sum() / y_val.shape[0]  # 返回准确率
```

4.9

使用书中式4.9、4.10、4.11，可以将原书中4.5式可以推广为：

$$Gini(D) = 1 - \sum_{k=1}^{|y|} (\tilde{p}_k)^2$$

属性a的基尼指数可以推广为：

$$Gini\_index(D, a) = p \times Gini\_index(\tilde{D}, a) = p \times \sum_{v=1}^{V} \tilde{v} Gini(D^v)$$