机器学习基础 实验三 实验报告

学号：202122202214

姓名：马贵亮

班级：2021级软件5班

实验目的

本实验通过鸢尾花数据集iris.csv来实现对决策树进一步的了解。其中，Iris鸢尾花数据集是一个经典数据集，在统计学习和机器学习领域都经常被用作示例。数据集内包含3类共150条记录，每类各50个数据，每条记录都有4项特征：花萼长度、花萼宽度、花瓣长度、花瓣宽度，可以通过这4个特征预测鸢尾花卉属于（iris-setosa, iris-versicolour, iris-virginica）三个类别中的哪一品种。

实验内容

本实验将五分之四的数据集作为训练集对决策树模型进行训练；将剩余五分之一的数据集作为测试集，采用训练好的决策树模型对其进行预测。训练集与测试集的数据随机选取。本实验采用准确率(accuracy)作为模型的评估函数：预测结果正确的数量占样本总数。

$$Acc = \frac{TP + TN}{TP + TN + FP + FN}$$

实验要求：

1.本实验要求输出测试集各样本的预测标签和真实标签，并计算模型准确率。另外，给出3个可视化预测结果。

2.决策树算法可以分别尝试ID3, C4.5, CART树，并评判效果。

3.（选做）：对你的决策树模型进行预剪枝与后剪枝

4.（选做）：分别做C4.5 和 CART 树的剪枝并比较不同。

注意事项：

编程语言不限，推荐使用python；决策树模型需要自己实现，不可使用已有的第三方库；实验报告中提供的代码需要有必要的注释。

实验过程

1. 数据分析

本次实验我们要处理的是iris数据集，该数据集的大致形式结构如下：

SepalLength	SepalWidth	PetalLength	PetalWidth	Species(标签)

其中前四列中SepalLength、SepalWidth、PetalLength、PetalWidth为其数据属性，Species为其数据标签，数据标签共有三种，分别为：**iris-setosa, iris-versicolour, iris-virginica。**数据集内包含3类共150条记录，每类各50个数据，每条记录都有4项特征：花萼长度、花萼宽度、花瓣长度、花瓣宽度。

特征分布列表：

属性名	最小值	最大值	均值	方差
SepalLength	4.3	7.9	5.843	0.686
SepalWidth	2	4.4	3.054	0.188
PetalLength	1	6.9	3.759	3.113
PetalWidth	0.1	2.5	1.199	0.582

由于我们采用决策树算法，因此对于数值型的变量其实并不需要特殊的处理，而本数据集中除去标签也不存在文本类属性，因此我们可以粗暴的读取数据并利用决策树算法（ID3、C4.5、CART）算法来构造决策树并且来进行预测与测试。

2. 数据分割

实验目的中阐述到，本实验将五分之四的数据集作为训练集对决策树模型进行训练；将剩余五分之一的数据集作为测试集，采用训练好的决策树模型对其进行预测。因此需要将数据进行分割，分割成为120个数据样本的训练集和30个数据样本的测试集，我们大致观测一下数据集，这个数据集中所有相同标签的样本集中在一起，因此我们在进行数据分割之前必须将数据进行打乱，由于本实验要求代码自行实现。因此我编写如下函数来对数据进行切割：

```
import numpy as np

# 数据分割，data原始数据，test_size测试集占比，target标签栏，random_state打乱的随机种子
def splitForData(data, test_size, target, random_state=42):
    np.random.seed(random_state)
    data_shuffled = data.sample(frac=1).reset_index(drop=True)
    train_size = int((1 - test_size) * len(data))
    train_data = data_shuffled[:train_size]
    test_data = data_shuffled[train_size:]
    X_train = train_data.drop(target, axis=1)
    y_train = train_data[target]
    X_test = test_data.drop(target, axis=1)
    y_test = test_data[target]
    return X_train, y_train, X_test, y_test
```

3.离散值ID3决策树实现

3.1 ID3Discrete构建

根据最简单的决策树，即只能处理离散值的ID3决策树，其对于每个分支节点的分支希望据册数的分支节点所包含的样本尽可能属于同一类别，即节点的纯度(purity)越来越高，ID3采用的纯度为信息增益(information gain)。

假定当前样本集合 $\mathbf{D}$ 中第k 类样本所占比例比例为p_k，则$\mathbf{D}$ 的信息熵(information entropy)为

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

$Ent(\mathbf{D})$ 越小， $\mathbf{D}$ 的纯度越高。

假定离散属性 a 有 V 个可能取值 $\{a_1, a_2, \ldots, a_V\}$，若采用$a$ 来对样本集 $\mathbf{D}$ 进行划分，则会产生 V 个分支节点。其中第v个分支节点包含了$\mathbf{D}$ 中所有在属性a 上取值为a_v 的样本，记作$\mathbf{D}_v$。因此可以求得$\mathbf{D}_v$的信息熵，考虑到不同分支节点所包含的样本数不同，因此分解结点赋予权重$|\mathbf{D}_v|/|\mathbf{D}|$，即所包含的分支节点越多影响越大。因此可以求得基于该属性a分类的信息增益：

$$Gain(\mathbf{D}; a) = Ent(\mathbf{D}) - \sum_{v=1}^{V} \frac{|\mathbf{D}_v|}{|\mathbf{D}|} Ent(\mathbf{D}_v)$$

信息增益率越大，意味着使用属性a 来进行划分所获得的纯度提升越大，因此在每个节点进行分支时所选取的属性 a 由下式确定：

$$a_* = \underset{a \in A}{\mathbf{argmax}} Gain(D; a)$$

随着ID3决策树每次进行节点分裂，顺便删除该点包含数据中该最优属性，随着最大深度或者最小划分数（预剪枝）或者所以节点都是叶子节点时，结束决策树的创建。采用递归建树即可实现如上过程。因此我设计了完全针对离散值的 `ID3Discrete.py` 的代码，来实现基于完全离散值考虑的ID3数来预测iris数据集。

3.2 代码实现

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from SplitData import splitForData
import pydotplus
from graphviz import Digraph
from IPython.display import Image, display


def entropy(y):
    labels = np.unique(y)
    Ent = 0.0
    for label in labels:
        p = len(y[y == label]) / len(y)
        Ent -= p * np.log(p)
    return Ent


class ID3Discrete:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None
        self.result = None # 存储离散值的标签

    def best_split(self, X, y):
        base_entropy = entropy(y)
        best_information_gain = -1
        best_feature = None
        best_splits = None

        for i in range(X.shape[1]):
            values = X[:, i]
            unique_values = np.unique(values)
            splits = {}
            for value in unique_values:
                splits[value] = y[values == value]

            new_entropy = 0
            for subset in splits.values():
                new_entropy += len(subset) / len(y) * entropy(subset)

            information_gain = base_entropy - new_entropy

            if information_gain > best_information_gain:
                best_information_gain = information_gain
                best_feature = i
                best_splits = splits

        return best_feature, best_splits,best_information_gain

```

```python
    def build_tree(self, X, y, features, depth=0):
        tree = {}
        if len(np.unique(y)) == 1:
            tree['type'] = 'leaf'
            tree['value'] = self.result[np.unique(y)[0]]
            return tree
        if len(features) == 0 or (self.max_depth is not None and depth >= self.max_depth):
            tree['type'] = 'leaf'
            tree['value'] = self.result[np.bincount(y).argmax()]
            return tree

        best_feature, best_splits,best_gain = self.best_split(X, y)
        if best_feature is None:
            tree['type'] = 'leaf'
            tree['value'] = self.result[np.bincount(y).argmax()]
            return tree

        # tree = {features[best_feature]: {}}
        tree['best_feature'] = features[best_feature]
        tree['splits'] = {}
        tree['type'] = "inner"
        tree['gain'] = best_gain
        remaining_features = features[:best_feature] + features[best_feature + 1:]

        for feature_value, subset in best_splits.items():
            subset_X = X[X[:, best_feature] == feature_value]
            subset_y = subset
            subset_X = np.delete(subset_X, best_feature, axis=1)

            subtree = self.build_tree(subset_X, subset_y, remaining_features, depth + 1)
            tree['splits'][feature_value] = subtree
        return tree

    def fit(self, X, y):
        features = X.columns.tolist()
        X = X.values
        unique_labels = pd.unique(y)  # 获取y中的唯一值
        self.result = list(unique_labels)  # 保存标签列表

        # 转换y为索引值
        label_to_index = {label: index for index, label in enumerate(unique_labels)}  # 创建从标签到索引的映射
        y_indexed = y.map(label_to_index)  # 将所有y值替换为对应的索引
        self.tree = self.build_tree(X, y_indexed, features, 0)

    def predict_one(self, x, tree):
        if tree['type'] == 'leaf':
            return tree['value']
        feature = tree['best_feature']
        feature_value = x[feature]
        if feature_value in tree['splits']:
            return self.predict_one(x, tree['splits'][feature_value])
        else:
            return self.result[0]

```

```
    def predict(self, X):
        return X.apply(lambda x: self.predict_one(x, self.tree), axis=1)
```

在建树过程中对于每个节点用一个字典来表示其之间的关系以及自身的属性。

对于内部节点的字典结构如下：（实际过程中中间过程均用 `tree` 作为变量名）

```
node = {
    'best_feature':xxx
    'splits':{split_val:subtree}
    'type':'inner'
    'gain':xxx
}
```

对于叶子节点的字典结构如下：（实际过程中中间过程均用 `tree` 作为变量名）

```
node={
    'type':'leaf'
    'value':xxx
}
```

而在 `predict` 中对于该元素查不到与自身所有属性所对应的路径与节点时，直接返回第一类的标签作为预测结果。

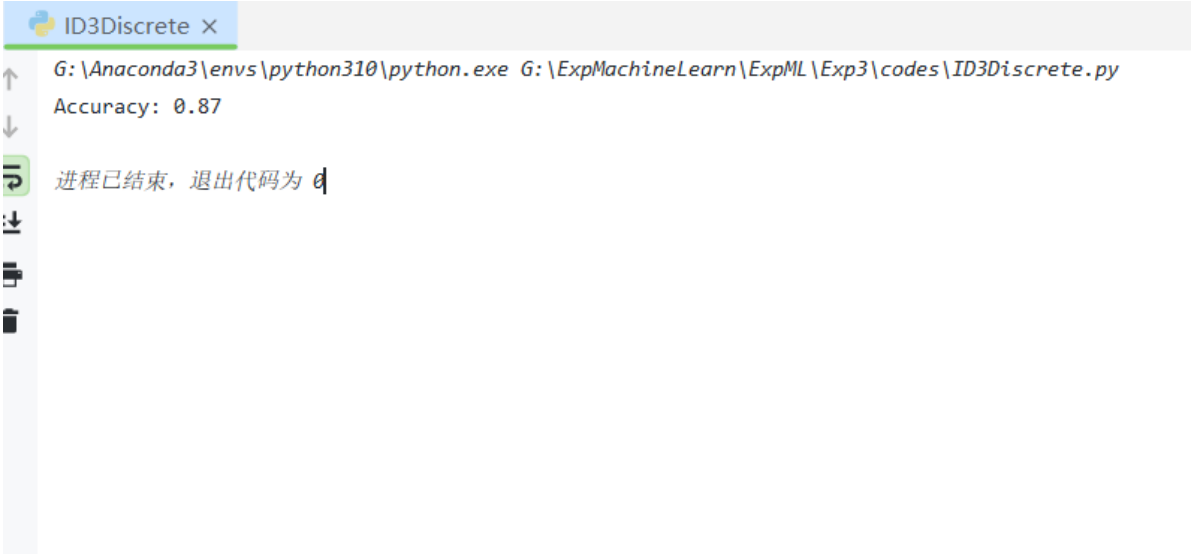
3.3 测试集预测结果

通过上述代码的实现，我实现了一个简单的只能处理离散值，并且把Iris数据集完全看作离散属性处理的决策树，我们对其用测试集进行预测，查看其预测的效果。

编写主函数，并且测试其在测试集上的准确率：

```
if __name__ == "__main__":
    df = pd.read_csv('../data/iris.csv')
    X_train, y_train, X_test, y_test = splitForData(df, 0.2, 'Species', 42)
    model = ID3Discrete(max_depth=5)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    predictions = np.array(predictions)
    accuracy = np.mean([predictions[i] == y_test.iloc[i] for i in
range(len(y_test))])
    print(f"Accuracy: {accuracy:.2f}")
```

测试结果如下：



可以发现结果在87%，可以发现即使是全部考虑成为离散的变量其也存在较高的准确率，这说明了ID3算法的优越性。

3.4 ID3离散决策树可视化展示

通过本地下载安装`graphviz`应用，并且在函数中调用`from graphviz import Digraph`，即可通过遍历来将整个树绘制出来，其中提供，假定初始化为`dot = Digraph()`，则有`dot.node(node_name)`、`dot.edge(node_name1,node_name2)`来包含顶点和边，整个系统由`node_name`来声明和定义节点。并且编写下列树的遍历的代码来实现树的绘制。（后续的树的绘制代码均和该代码类似，在此不在阐述）

```
    def plot_tree(self):

        def add_nodes_edges(tree, dot=None, parent_name=None, edge_label=None, feature_name=None):
            if dot is None:
                dot = Digraph()
                # dot = Digraph(graph_attr={"splines": "line"})

            if tree['type'] == 'leaf':
                if edge_label is not None:
                    node_name = f"{feature_name} = {edge_label}\n{str(tree['value'])}"
                else:
                    node_name = str(tree['value'])
                dot.node(node_name, color="red")
                if parent_name:
                    dot.edge(parent_name, node_name)
            else:
                if edge_label:
                    node_name = f"{feature_name} = {edge_label}\nDivided By {tree['best_feature']}\nGain={round(tree['gain'],4)}"
                else:
                    node_name = f"Root\nDivided By {tree['best_feature']}\nGain={round(tree['gain'],4)}"
                dot.node(node_name, color="blue", shape='box')
                if parent_name:
                    dot.edge(parent_name, node_name)

                for feature_value, subtree in tree['splits'].items():
```

```
                    add_nodes_edges(subtree, dot=dot, parent_name=node_name,
edge_label=str(feature_value),
                                    feature_name=tree['best_feature'])

            return dot

        dot = add_nodes_edges(self.tree)
        return dot
```

完成遍历后我们得到一个`dot`，需要将这个对象的内容进行保存并且渲染成为图片格式才能更好的展示树的形状，因此在主函数中增加下述代码：

```
    dot = model.plot_tree()
    dot.render("../pic/ID3/ID3discrete", format='png')
    image = Image(filename="../pic/ID3/ID3discrete.png")
    display(image)
```

即可保存成为png图形格式，在执行后会产生两个文件，无后缀文件用于保存对应的`dot`中的内容，.png文件用来保存渲染后的图片。如下：

可以发现由于该决策树对连续变量完全采用离散的方法进行处理，因此这颗决策树的宽度非常的宽，在此展示效果并不理想，该图片会作为附件，一起作为实验报告提交。

由于考虑到用纯离散处理的决策树图片太过肥胖臃肿，并且在正常情况下不会采用这样的方式处理连续值，因此不再用该离散树展示三个可视化预测结果（树上搜索图）。

4. 连续值ID3决策树实现

4.1 ID3Continous构建

在我们所授课的内容中，许老师向我们讲到ID3的一大缺点就是不能处理连续值，可是我们又在C4.5中学到对于一个连续值的属性可以对其进行分箱，分箱之后便是连续属性。连续值比较优秀的分箱其实是一个比较麻烦的过程，尤其是在处理决策树分支的过程中。但是可以通过C4.5或者CART中处理方法的思想，即每次分支对一个属性进行大于小于某个阈值的二分法，并且保留该属性不像离散值一样删去，如果该那么当这个属性被多次切割时，这个过程就完成了分箱。事实上这也是对于连续值有监督分箱的一种基本思想。

那么对于每次分支过程，所执行的操作即为：

```
对于该节点中所有的样本，遍历每一个属性a：
    遍历该属性所有取值v：
        小于等于v为左儿子节点
        大于等于v为右儿子节点
        计算通过该属性该取值分支的信息增益的大小
        记录最大信息增益对应的属性为最佳属性
        记录最大信息增益对应的分界点为阈值
```

通过这样的操作即可实现，基于二分类的连续值处理的ID3决策树。

4.2 代码实现

```
class ID3Continuous:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None
        self.result = None

    def entropy(self, y):
        labels = np.unique(y)
        Ent = 0.0
        for label in labels:
            p = len(y[y == label]) / len(y)
            Ent -= p * np.log(p)
        return Ent

    def best_spilt(self, X, y):
        base_entropy = self.entropy(y)
        best_information_gain = -1
        best_threshold = None
        best_feature = None

        for i in range(X.shape[1]):
            values = X[:, i]
            unique_values = np.unique(values)
            for threshold in unique_values:
                y_left = y[values <= threshold]
                y_right = y[values > threshold]
                p_left = len(y_left) / len(values)
                p_right = len(y_right) / len(values)

                information_gain = base_entropy - (p_left *
self.entropy(y_left) + p_right * self.entropy(y_right))
                if information_gain > best_information_gain:
                    best_information_gain = information_gain
                    best_threshold = threshold
                    best_feature = i
        return best_feature, best_threshold, best_information_gain

    def build_tree(self, X, y, features, depth=0):
        if len(np.unique(y)) == 1:
            return self.result[np.unique(y)[0]]

        if self.max_depth is not None and depth > self.max_depth:
            return self.result[np.bincount(y).argmax()]

        best_feature, best_threshold,best_gain = self.best_spilt(X, y)
        left_indices = X[:, best_feature] <= best_threshold
        right_indices = X[:, best_feature] > best_threshold
        node = {}
        node['feature'] = features[best_feature]
        node['threshold'] = best_threshold
        node['gain'] = best_gain
        node['left'] = self.build_tree(X[left_indices], y[left_indices],
features, depth + 1)
        node['right'] = self.build_tree(X[right_indices], y[right_indices],
features, depth + 1)
```

```
        return node

    def fit(self, X, y):
        features = X.columns.tolist()
        X = X.values
        unique_labels = pd.unique(y)  # 获取y中的唯一值
        self.result = list(unique_labels)  # 保存标签列表

        # 转换y为索引值
        label_to_index = {label: index for index, label in
enumerate(unique_labels)}  # 创建从标签到索引的映射
        y_indexed = y.map(label_to_index)  # 将所有y值替换为对应的索引
        self.tree = self.build_tree(X, y_indexed, features, 0)

    def predict(self, X):
        predictions = []
        for index, sample in X.iterrows():
            node = self.tree
            while isinstance(node, dict):
                if sample[node['feature']] <= node['threshold']:
                    node = node['left']
                else:
                    node = node['right']
            predictions.append(node)
        return predictions
```

其整体的代码逻辑和上述的完全离散值的ID3基本保持一致，但是部分处理略有不同，一是加入了刚刚阐述的二分选取阈值的实体代码，二是对于节点字典的处理发生了变化。（对于第二点实际原因是因为我先写的处理连续值的代码，在许老师授课相关内容后又实现了一个完全基于离散值的离散决策树。）

新的节点处理如下：

内部节点：

```
node={
    'feature':分支所用的特征
    'threshold':分支所用的阈值
    'gain':
    'left':左子树
    'right':右子树
}
```

叶子节点：

```
value(即对应的标签值)
```

而在`predict`中对于该元素查不到与自身所有属性所对应的路径与节点时，直接返回第一类的标签作为预测结果。

4.3 测试集预测结果

编写主函数代码，对测试集进行预测，得到其预测的结果准确率

```
if __name__ == '__main__':
    df = pd.read_csv('../data/iris.csv')
    X_train, y_train, X_test, y_test = splitForData(df, 0.2, 'Species')
    model = ID3Continuous(max_depth=10)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    print(predictions)
    accuracy = np.mean([predictions[i] == y_test.iloc[i] for i in range(len(y_test))])
    print(f"Accuracy: {accuracy:.2f}")
```

运行后得到模型的准确率：93%，与只用离散值处理的有了较大的提高。

4.4 ID3连续决策树可视化展示（三个预测过程展示）

类似3.4中所实现的内容，利用`graphviz`来实现决策树的可视化展示，依旧采用递归建树，建树代码如下：

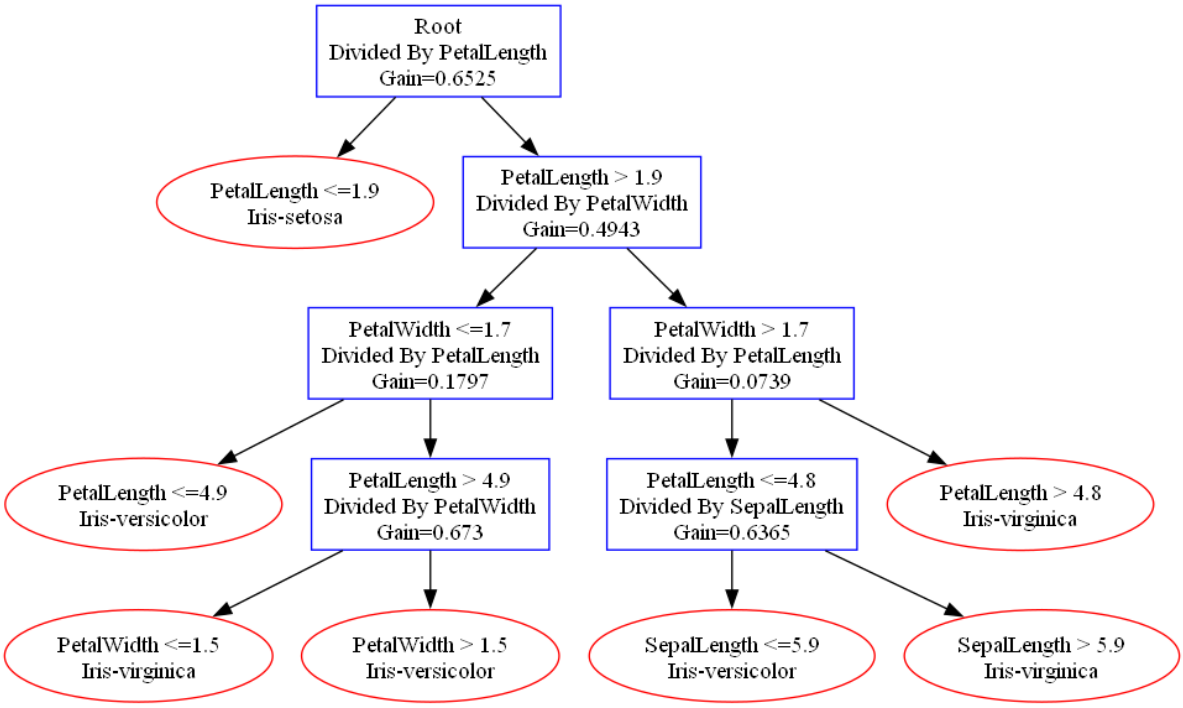
```
    def plot_tree(self):
        def add_nodes_edges(tree, dot=None, parent_name=None, edge_label=None, feature_name=None):
            if dot is None:
                dot = Digraph()
                # dot = Digraph(graph_attr={"splines": "line"})

            if not isinstance(tree, dict):
                if edge_label is not None:
                    node_name = f"{feature_name} {edge_label}\n{str(tree)}"
                else:
                    node_name = str(tree)
                dot.node(node_name, color="red")
                if parent_name:
                    dot.edge(parent_name, node_name)
            else:
                if edge_label:
                    node_name = f"{feature_name} {edge_label}\nDivided By {tree['feature']}\nGain={round(tree['gain'],4)}"
                else:
                    node_name = f"Root\nDivided By {tree['feature']}\nGain={round(tree['gain'],4)}"
                dot.node(node_name, color="blue", shape='box')
                if parent_name:
                    dot.edge(parent_name, node_name)

                if 'left' in tree:
                    add_nodes_edges(tree['left'], dot=dot, parent_name=node_name,
```

```
                                    edge_label="<=" +
str(tree['threshold']),
                                    feature_name=tree['feature'])
                if 'right' in tree:
                    add_nodes_edges(tree['right'], dot=dot,
parent_name=node_name,
                                    edge_label="> " +
str(tree['threshold']),
                                    feature_name=tree['feature'])
            return dot

        dot = add_nodes_edges(self.tree)
        return dot
```

主函数中增加：

```
    dot = model.plot_tree()
    dot.render("../pic/ID3/ID3continous", format='png')
    image = Image(filename="../pic/ID3/ID3continous.png")
    display(image)
```

运行并渲染后的结果如下：



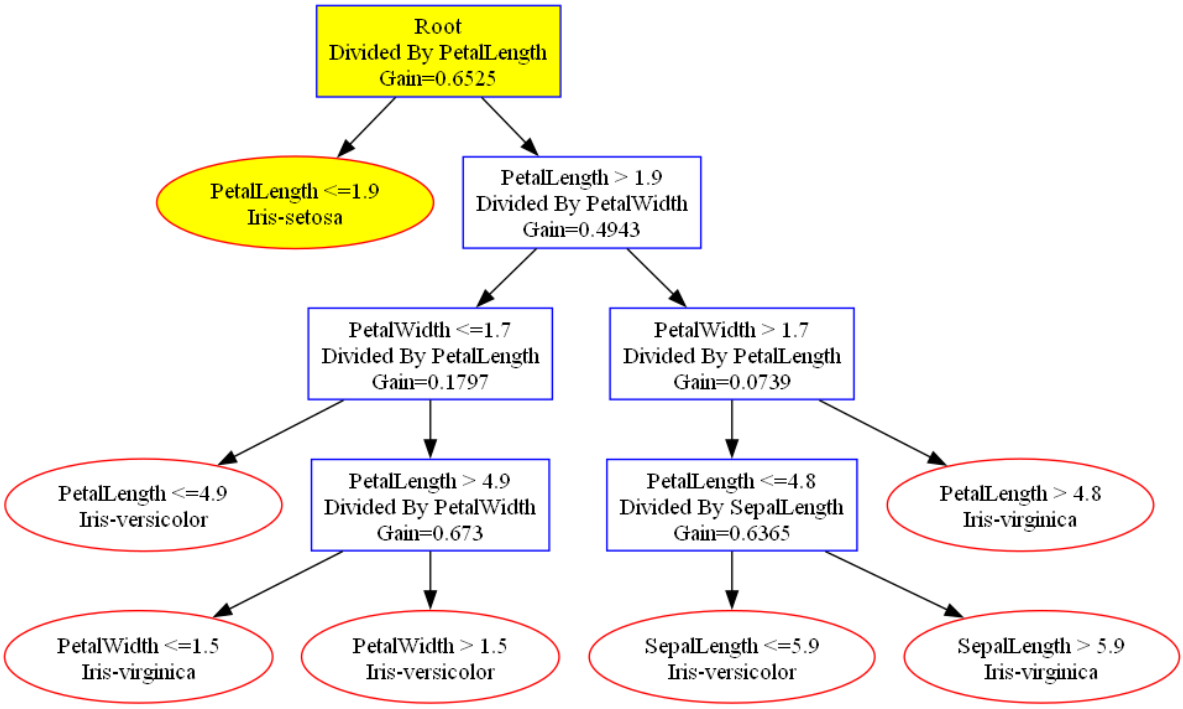
可以非常清晰的看到决策树的整体形状，相较于完全离散值处理的决策树，整个树的结构变得更加简单，更加客观。整体的泛化能力也得到了提升。为了更好的展示该决策树的预测过程，我选用以下三个数据来可视化展示决策树的预测过程。

数据一：数据集中index=10的元素

数据：

```
SepalLength          5.4
SepalWidth           3.7
PetalLength          1.5
PetalWidth           0.2
```

搜索过程

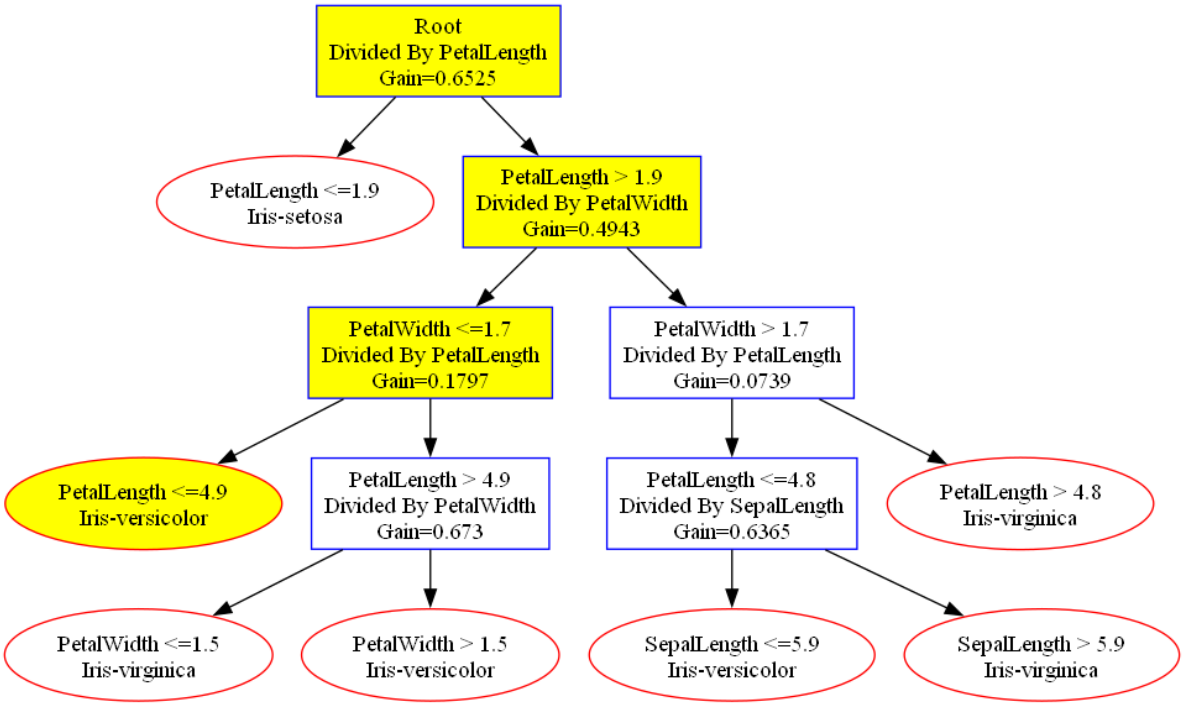


数据二：数据集中index=88的元素

数据：

```
SepalLength             5.6
SepalWidth              3.0
PetalLength             4.1
PetalWidth              1.3
```

搜索过程

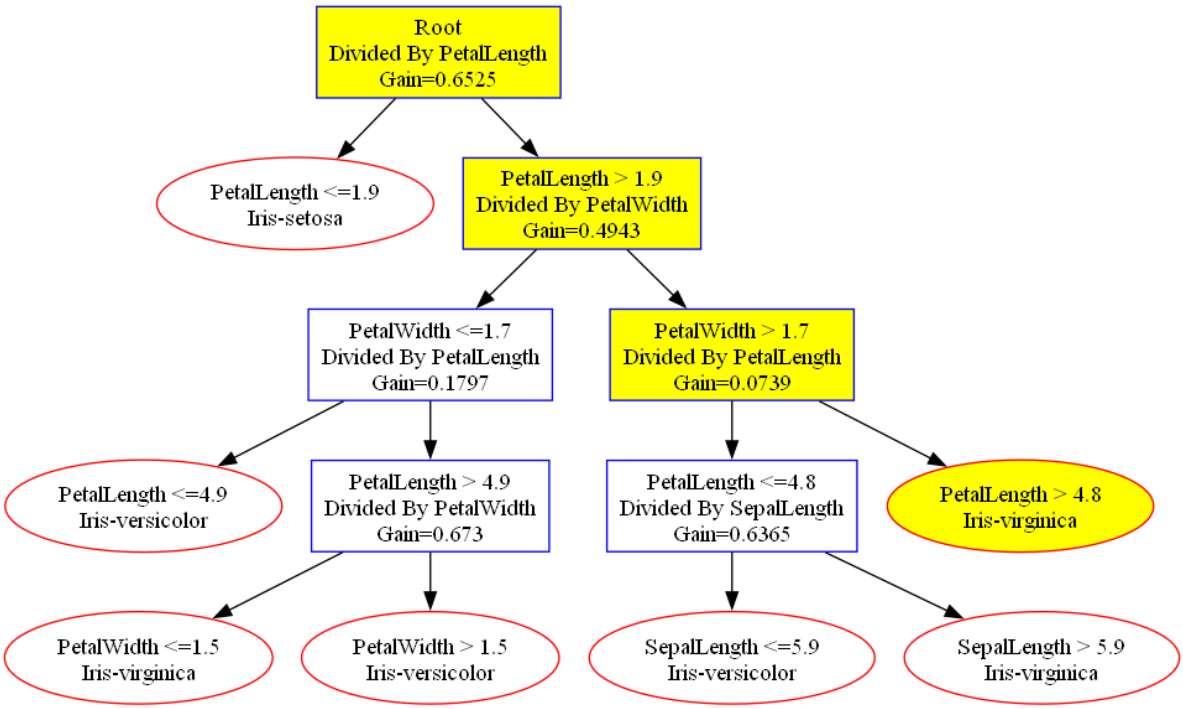


数据三：数据集中index=120的元素

数据：

```
SepalLength             6.9
SepalWidth              3.2
PetalLength             5.7
PetalWidth              2.3
```

搜索过程



5. C4.5决策树实现

5.1 C4.5决策树的构建

事实上，信息增益准则对可取值数目较多的属性有所偏好，为了减少这种偏好所带来的不利影响，C4.5算法采用增益率(gain ratio) 开选择最优划分属性，信息增益率定义为：

$$Gain\_ratio(\mathbf{D};a) = \frac{Gain(\mathbf{D};a)}{IV(a)}$$

$$IV(a) = -\sum_{v=1}^{V} \frac{|\mathbf{D}_v|}{|\mathbf{D}|} log_2 \frac{|\mathbf{D}_v|}{|\mathbf{D}|}$$

增益率准则对可取数值较小的属性有所偏好，因此C4.5算法并不是直接选取增益率最大的候选划分属性，而是使用了一种启发式算法：先从候选划分属性中找出信息增益高于平均水平的属性，再从其中选择增益率最高的。基于这样的过程，我们可以设计C4.5的代码。

5.2 代码实现

```python
class C45:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None
        self.result = None

    def fit(self, X, y):
        self.features = X.columns.tolist()
        # 创建从类别标签到索引的映射，并保存原始标签至self.result
        unique_labels = pd.unique(y)  # 获取y中的唯一值
```

```
        self.result = list(unique_labels)  # 保存标签列表

        # 转换y为索引值
        label_to_index = {label: index for index, label in
enumerate(unique_labels)}  # 创建从标签到索引的映射
        y_indexed = y.map(label_to_index)  # 将所有y值替换为对应的索引

        # 构建决策树
        self.tree = self.build_tree(X, y_indexed, 1)

    def build_tree(self, X, y, depth):
        if len(np.unique(y)) == 1:
            return self.result[np.unique(y)[0]]

        if self.max_depth and depth > self.max_depth:
            return self.result[np.bincount(y).argmax()]

        best_feature, best_threshold, best_gain = self.best_split(X, y)
        if best_gain == 0:
            return np.bincount(y).argmax()

        left_X, right_X, left_y, right_y = self.split(X, y, best_feature,
best_threshold)

        node = {'feature': best_feature, 'threshold': best_threshold}
        node['left'] = self.build_tree(left_X, left_y, depth + 1)
        node['right'] = self.build_tree(right_X, right_y, depth + 1)
        node['gain_ratio'] = best_gain

        return node

    def best_split(self, X, y):
        best_gain = 0
        best_feature = None
        best_threshold = None

        for feature in X.columns:
            thresholds, gains = self.information_gain(X[feature], y)
            max_gain_idx = np.argmax(gains)
            if gains[max_gain_idx] > best_gain:
                best_gain = gains[max_gain_idx]
                best_feature = feature
                best_threshold = thresholds[max_gain_idx]

        return best_feature, best_threshold, best_gain

    def information_gain(self, X_feature, y):
        thresholds = np.unique(X_feature)
        gains = np.zeros(thresholds.shape)

        for i, threshold in enumerate(thresholds):
            left_y = y[X_feature <= threshold]
            right_y = y[X_feature > threshold]
            gains[i] = self.information_gain_ratio(y, left_y, right_y)

        return thresholds, gains

    def entropy(self, y):
```

```
        _, counts = np.unique(y, return_counts=True)
        probabilities = counts / counts.sum()
        return -np.sum(probabilities * np.log2(probabilities))

    def information_gain_ratio(self, y, left_y, right_y):
        parent_entropy = self.entropy(y)
        left_entropy = self.entropy(left_y)
        right_entropy = self.entropy(right_y)
        p_left = len(left_y) / len(y)
        p_right = len(right_y) / len(y)
        gain = parent_entropy - (p_left * left_entropy + p_right *
right_entropy)
        split_info = - (p_left * np.log2(p_left + 1e-10) + p_right *
np.log2(p_right + 1e-10))
        return gain / split_info if split_info > 1e-10 else 0

    def split(self, X, y, feature, threshold):
        left_mask = X[feature] <= threshold
        right_mask = X[feature] > threshold
        return X[left_mask], X[right_mask], y[left_mask], y[right_mask]

    def predict(self, X):
        predictions = []
        for index, sample in X.iterrows():
            node = self.tree
            while isinstance(node, dict):
                if sample[node['feature']] <= node['threshold']:
                    node = node['left']
                else:
                    node = node['right']
            predictions.append(node)
        return predictions
```

整体构建部分与上述两颗决策树区别不大，仅在最优特征选取上进行了修改，采用了信息增益率的方式。在节点字典设计上采用了和ID3Continous的方式一致的设计，预测部分对于不存在边界的处理仍旧填充第一类。

内部节点：

```
node={
    'feature':分支所用的特征
    'threshold':分支所用的阈值
    'gain_ratio':
    'left':左子树
    'right':右子树
}
```

叶子节点：

```
value(即对应的标签值)
```

5.3 测试集预测结果

编写主函数并且执行并获得预测结果

```
if __name__ == '__main__':
    df = pd.read_csv('../data/iris.csv')
    X_train, y_train, X_test, y_test = splitForData(df, 0.2, 'Species')
    model = C45(max_depth=10)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    print(predictions)
    accuracy = np.mean([predictions[i] == y_test.iloc[i] for i in
range(len(y_test))])
    print(f"Accuracy: {accuracy:.2f}")
```

执行后得到准确率为93%

5.4 C4.5可视化展示（三个预测过程展示）

此时数据分割的随机种子为默认即42

类似3.4中所实现的内容，利用`graphviz`来实现决策树的可视化展示，依旧采用递归建树，建树代码如下：

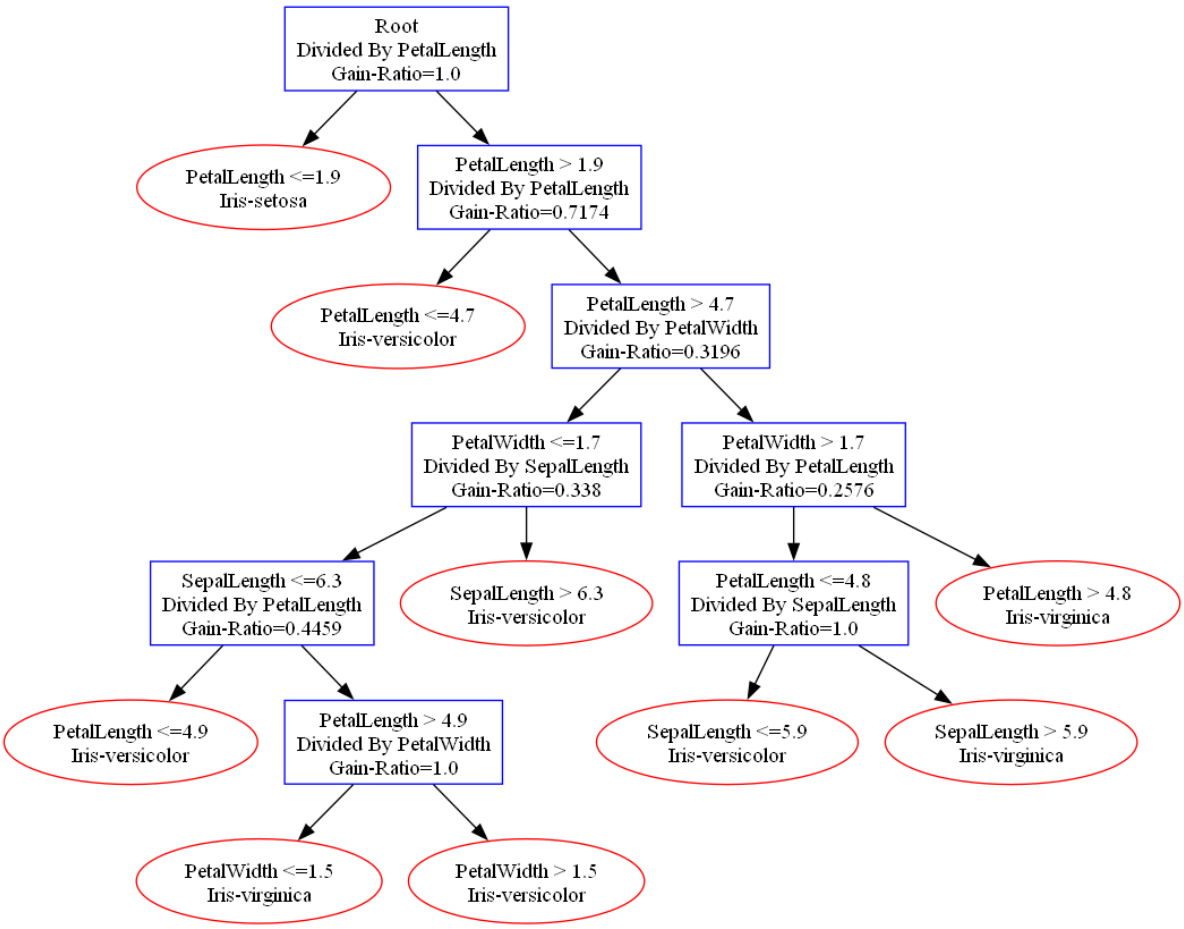
```
    def plot_tree(self):
        def add_nodes_edges(tree, dot=None, parent_name=None,
edge_label=None, feature_name=None):
            if dot is None:
                dot = Digraph()
                # dot = Digraph(graph_attr={"splines": "line"})

            if not isinstance(tree, dict):
                if edge_label is not None:
                    node_name = f"{feature_name} {edge_label}\n{str(tree)}"
                else:
                    node_name = str(tree)
                dot.node(node_name, color="red")
                if parent_name:
                    dot.edge(parent_name, node_name)
            else:
                if edge_label:
                    node_name = f"{feature_name} {edge_label}\nDivided By
{tree['feature']}\nGain-Ratio={round(tree['gain_ratio'], 4)}"
                else:
                    node_name = f"Root\nDivided By {tree['feature']}\nGain-
Ratio={round(tree['gain_ratio'], 4)}"
                dot.node(node_name, color="blue", shape='box')
                if parent_name:
```

```
                    dot.edge(parent_name, node_name)

                if 'left' in tree:
                    add_nodes_edges(tree['left'], dot=dot,
parent_name=node_name,
                                    edge_label="<=" +
str(tree['threshold']),
                                    feature_name=tree['feature'])
                if 'right' in tree:
                    add_nodes_edges(tree['right'], dot=dot,
parent_name=node_name,
                                    edge_label="> " +
str(tree['threshold']),
                                    feature_name=tree['feature'])
            return dot

        dot = add_nodes_edges(self.tree)
        return dot
```

主函数中增加：

```
    dot = model.plot_tree()
    dot.render("../pic/C45/3C45", format='png')
    image = Image(filename="../pic/C45/3C45.png")
    display(image)
```

运行并渲染后的结果如下：



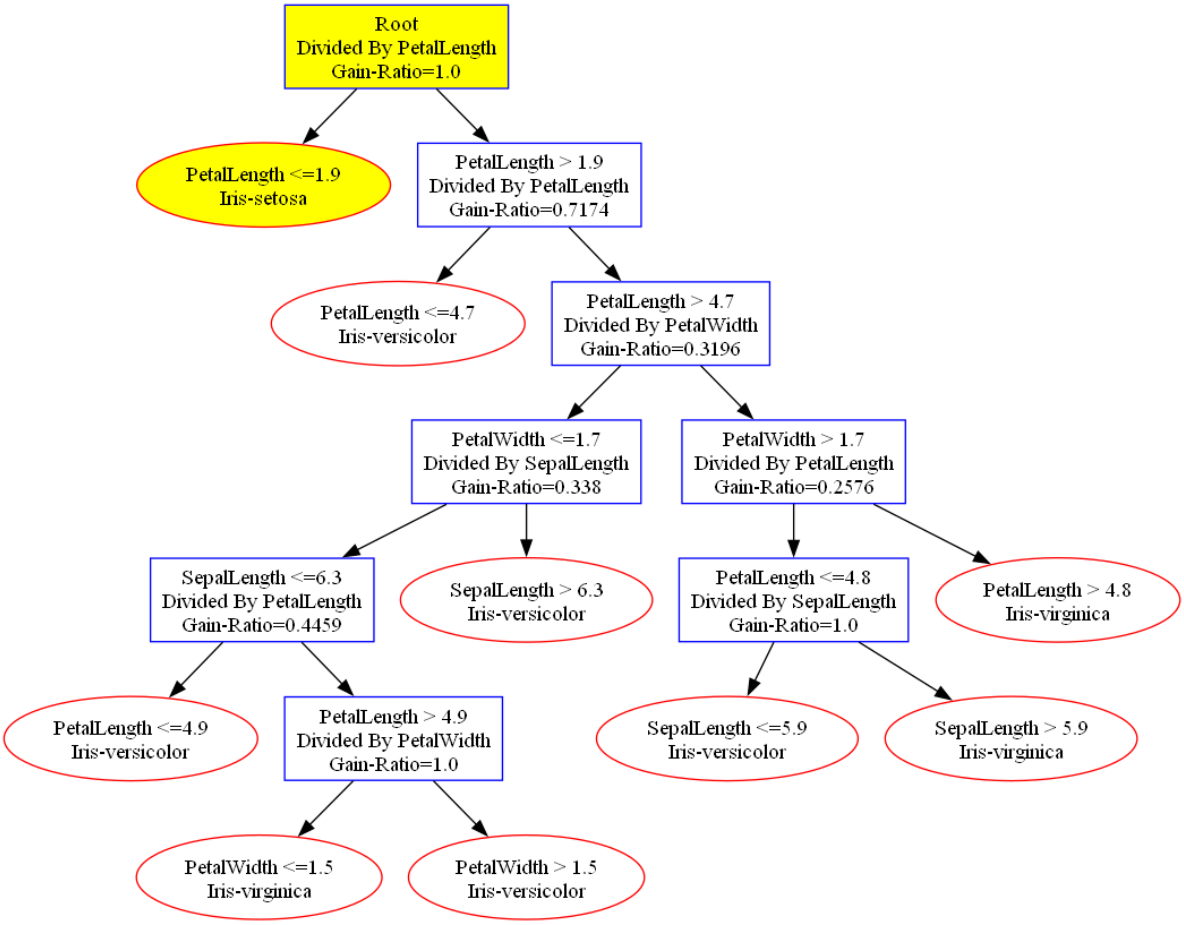
由于选取了一个造成树结构比较复杂的随机种子（测试分析的），整体树状结构尚可。

数据一：数据集中index=10的元素

数据：

```
SepalLength          5.4
SepalWidth           3.7
PetalLength          1.5
PetalWidth           0.2
```

搜索过程

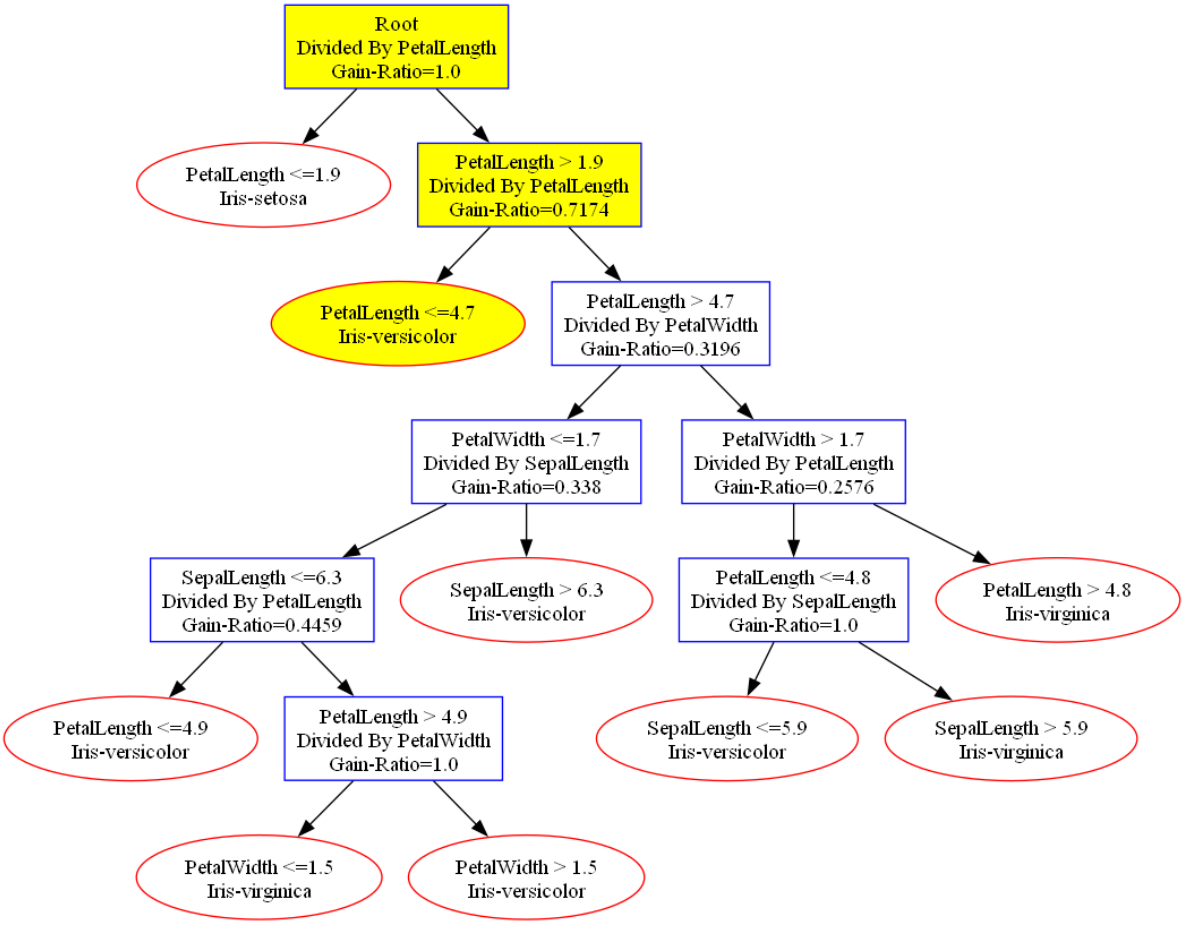


数据二：数据集中index=88的元素

数据：

```
SepalLength          5.6
SepalWidth           3.0
PetalLength          4.1
PetalWidth           1.3
```

搜索过程

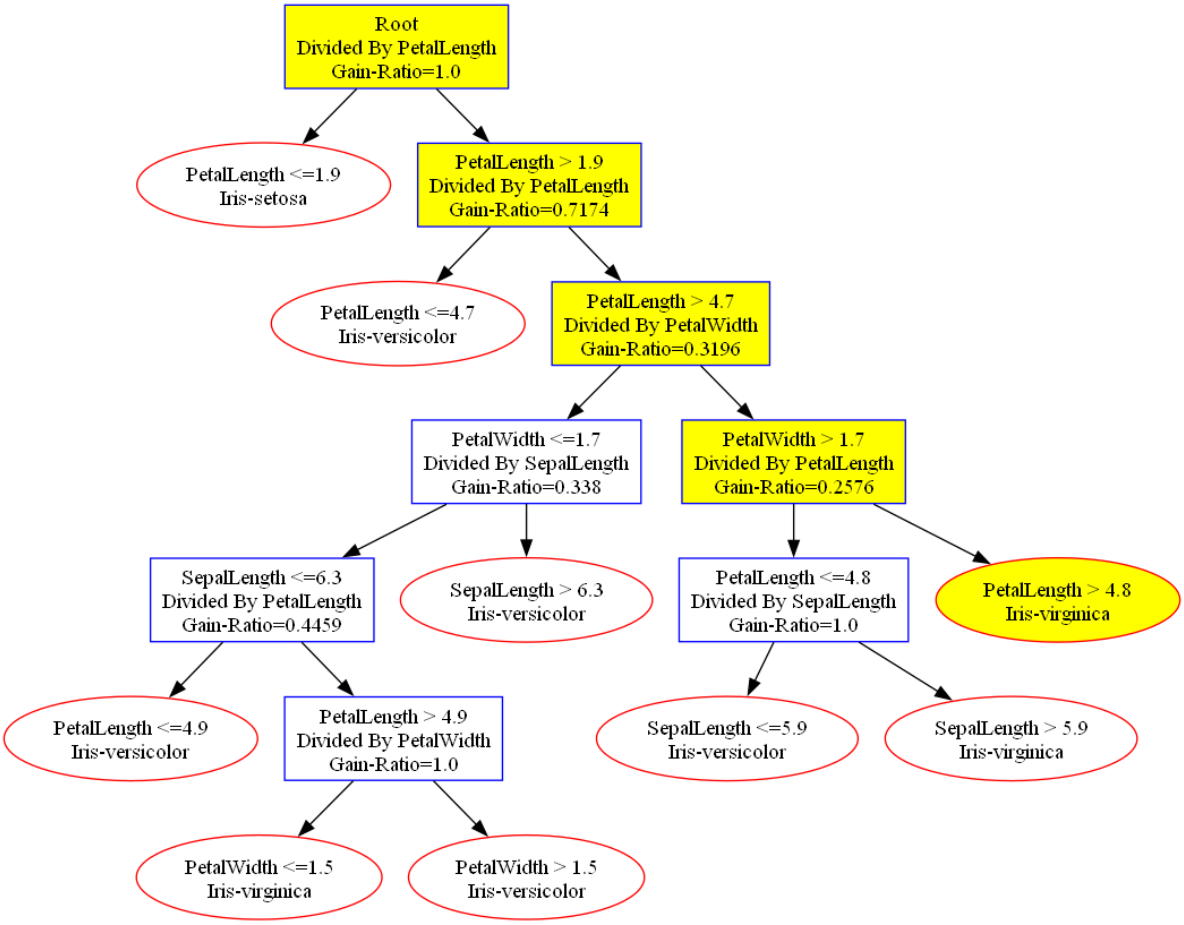


数据三：数据集中index=120的元素

数据：

```
SepalLength             6.9
SepalWidth              3.2
PetalLength             5.7
PetalWidth              2.3
```

搜索过程



6. C4.5剪枝优化

6.1 预剪枝

决策树预剪枝（Pre-pruning）的策略是为了防止决策树在训练过程中过拟合而采取的一种措施。通过在构建树的过程中施加一些限制条件，可以提前停止树的生长，从而避免生成过于复杂的树结构。以下是我从网络中寻找的几种常见的预剪枝策略。

1. **最大深度限制 (Max Depth)**

设置决策树的最大深度，限制树的层数。树的深度越大，模型的复杂度越高，容易过拟合。

实现方式： 在树的生成过程中，跟踪当前节点的深度，如果深度达到预设的最大深度，则不再继续分裂。

简单易行、可以有效控制树的复杂度、可能会过早停止分裂，导致欠拟合

2. **最小样本分割数 (Min Samples Split)**

设置一个节点在继续分裂之前必须包含的最小样本数。如果一个节点的样本数少于这个阈值，则不再分裂。

实现方式： 在每次分裂前检查当前节点的样本数，如果少于预设的最小样本分割数，则不再继续分裂。

可以防止小样本数的节点过度分裂、有助于避免过拟合、需要仔细调参，过高或过低的值都会影响模型性能

3. **最小样本叶节点数 (Min Samples Leaf)**

设置叶节点必须包含的最小样本数，确保每个叶节点至少有一定数量的样本。

实现方式： 在每次分裂后检查子节点的样本数，如果子节点的样本数少于预设的最小样本叶节点数，则不允许当前分裂。

保证每个叶节点有足够的样本进行估计、减少过拟合的风险、类似最小样本分割数，需要仔细调参

4. **最大叶节点数 (Max Leaf Nodes)**

限制决策树的叶节点数，不允许超过这个最大值。

实现方式： 在树生成过程中，跟踪当前的叶节点数，如果达到最大叶节点数，则停止分裂。

控制树的最终规模、简单易行、如果设置的最大叶节点数过低，可能导致欠拟合

5. **最大特征数 (Max Features)**

限制每次分裂时可用的最大特征数。通常用于随机森林算法中，但也可以单独应用于决策树。

实现方式： 在每次分裂时，随机选择一个预设数量的特征进行分裂，而不是使用所有特征。

增加模型的多样性、有助于减少过拟合、在决策树中单独使用时效果不如在随机森林中显著

6. **最小信息增益 (Min Impurity Decrease)**

设置一个阈值，只有当分裂后的信息增益大于这个阈值时才进行分裂。

实现方式： 在每次分裂前计算信息增益，如果增益小于预设的最小信息增益，则不进行分裂。

确保每次分裂都有显著的增益、防止不必要的分裂，减少过拟合、需要计算信息增益，增加计算复杂度

7. **基于统计检验的预剪枝**

使用统计检验（如卡方检验）来决定是否分裂节点。如果检验结果表明分裂后的效果不显著，则不进行分裂。

实现方式： 在每次分裂前进行统计检验，如果检验结果不显著（即 p 值大于预设阈值），则不进行分裂。

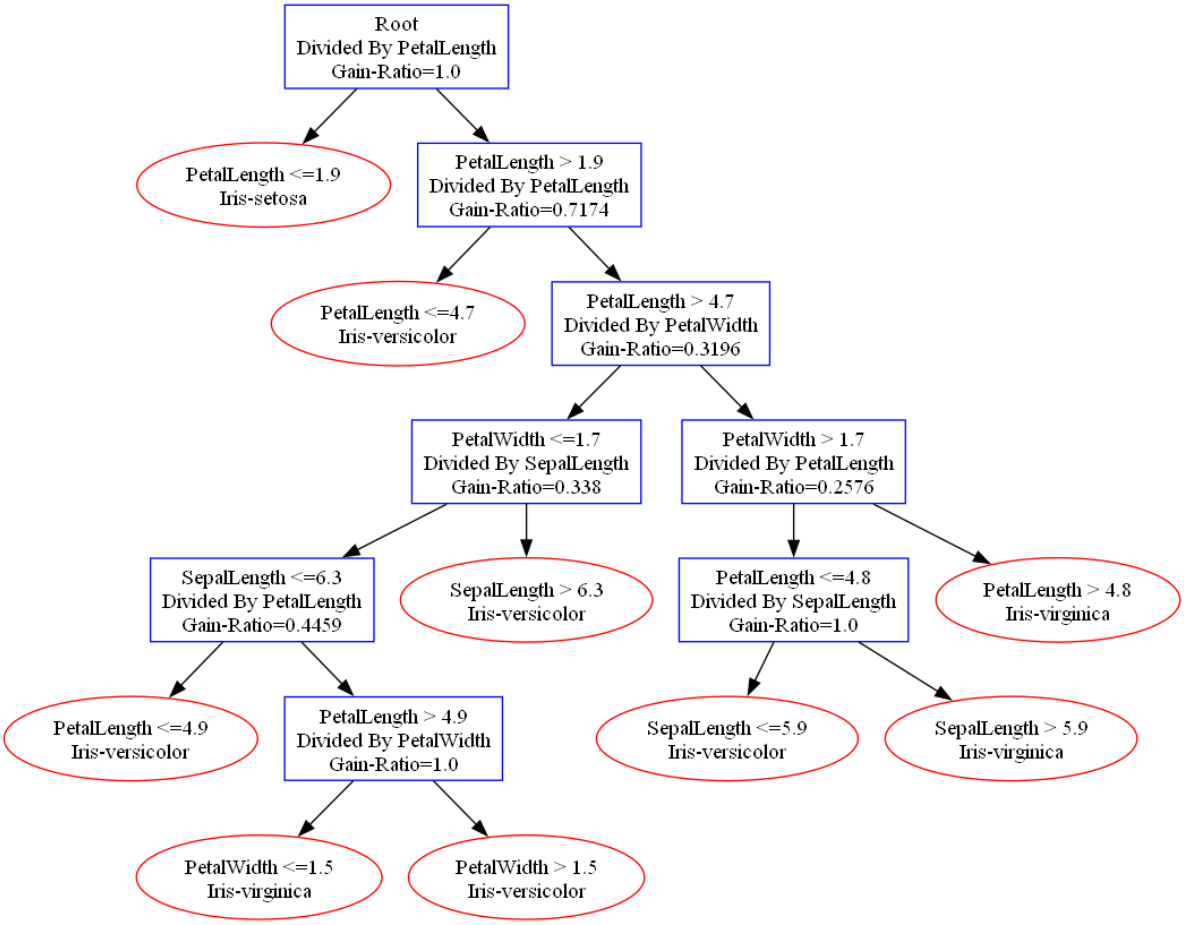
利用统计检验的结果，可以更加科学地决定是否分裂、计算复杂度较高、需要对统计检验结果有良好的理解和解释

在具体实现过程中，我先优先实现最大深度限制 (Max Depth)和最小样本分割数 (Min Samples Split)，控制这两者的过程比较简单，可以发现我在第五部分C4.5实现中已经放置了预剪枝的部分，即在建树过程中：

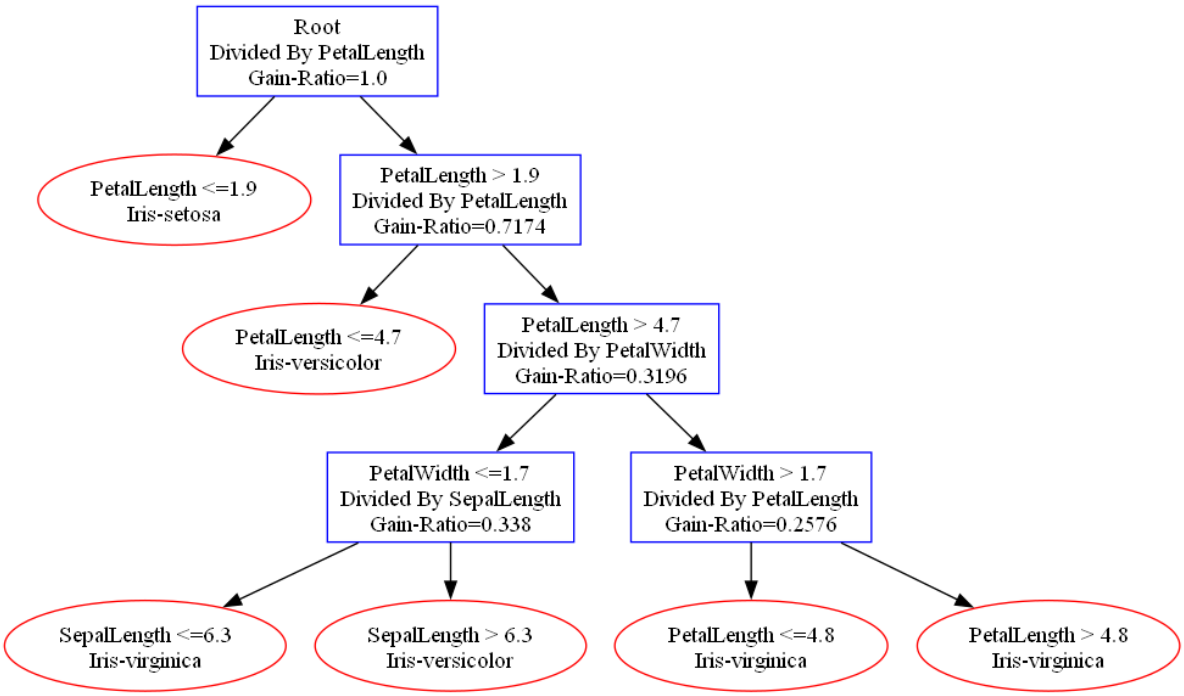
```
if self.max_depth and depth > self.max_depth:
    return self.result[np.bincount(y).argmax()]
if len(y) < self.min_samples_split:
    return self.result[np.bincount(y).argmax()]
```

基于第五部分以42的随机种子切割的C4.5决策树，maxdepth=4，min_samples_split=3可以得到如下两个数的区别：

原树：



预剪枝后的树：



6.2 后剪枝

后剪枝（Post-pruning）是在决策树构建完成后，对树进行剪枝，移除一些冗余的枝叶，从而简化模型。C4.5的后剪枝方法主要包括以下步骤：

1. **子树替换**：

   对于每一个非叶节点，考虑将其整个子树替换为一个叶节点。

   替换的标准是计算剪枝前后的误差，如果替换后误差降低或没有显著增加，则进行剪枝。

2. **子树提升**：

   这一步是对那些只有一个子节点的节点进行处理。

   如果某个节点只有一个子节点，则考虑将该节点提升，即用其子节点来替代它自身。

   这可以简化树的结构，同时不影响分类性能。

3. **计算误差**：

   使用验证集或交叉验证的方法来估计剪枝前后的误差。

   误差的计算可以基于统计学方法，例如用信赖区间估计错误率。

4. **实际剪枝**：

   根据上述步骤的误差计算结果，对树进行实际剪枝。

   通过替换或提升子树来减小树的复杂度。

为了实现这一部分的步骤，在`fit`的过程中进行简单修改，并且写入`prune`函数来进行尝试剪枝，代码如下：

```
class C45:
    # 初始化函数中省略部分代码

    def fit(self, X, y, isPruned=False, test_size=0.2, random_state=42):
        # 省略部分代码
        # 分离训练集和验证集
        X_train, X_val, y_train, y_val = self.train_val_split(X, y_indexed,
test_size, random_state)

        # 构建决策树
        self.tree = self.build_tree(X_train, y_train, 1)
        if isPruned:
            self.prune(self.tree, X_val, y_val)
        return self.evaluate_accuracy(X_val, y_val)

    def evaluate_accuracy(self, X, y):
        predictions = self.predict(X)
        accuracy = np.mean([predictions[i] == self.result[y.iloc[i]] for i
in range(len(y))])
        return accuracy

    def prune(self, tree, X_val, y_val):
        # 获取叶子节点并统计误差
        if tree['type'] == 'leaf':
            leaf_class = tree['class']
            val_accuracy = np.mean(y_val == leaf_class)
            return val_accuracy

        left_mask = X_val[tree['feature']] <= tree['threshold']
        right_mask = X_val[tree['feature']] > tree['threshold']
```

```
29
30          left_tree = tree['left']
31          right_tree = tree['right']
32
33          left_accuracy = self.prune(left_tree, X_val[left_mask],
    y_val[left_mask])
34          right_accuracy = self.prune(right_tree, X_val[right_mask],
    y_val[right_mask])
35
36          # 计算当前节点作为叶子节点的精度
37          majority_class = np.bincount(y_val).argmax()
38          majority_accuracy = np.mean(y_val == majority_class)
39
40          # 如果剪枝后精度更高，则将当前节点转换为叶子节点
41          if majority_accuracy >= (left_accuracy * np.mean(left_mask) +
    right_accuracy * np.mean(right_mask)):
42              tree['type'] = 'leaf'
43              tree['class'] = self.result[majority_class]
44              return majority_accuracy
45          else:
46              return left_accuracy * np.mean(left_mask) + right_accuracy *
    np.mean(right_mask)
47
```

这是一段全新的代码，大致的内容和原先的C4.5保持一致，这部分代码写的C45Inspire.py中，该部分的树内节点划分采用了如下格式：

内部节点：

```
1  node = {
2      'feature': best_feature,
3      'threshold': best_threshold,
4      'type': 'inner',
5      'left': self.build_tree(left_X, left_y, depth + 1),
6      'right': self.build_tree(right_X, right_y, depth + 1),
7      'gain_ratio': best_gain,
8      # most_label 是用于后剪枝过程中填充的使用
9      'most_label': self.result[np.bincount(y).argmax()]
10 }
```

叶子节点：

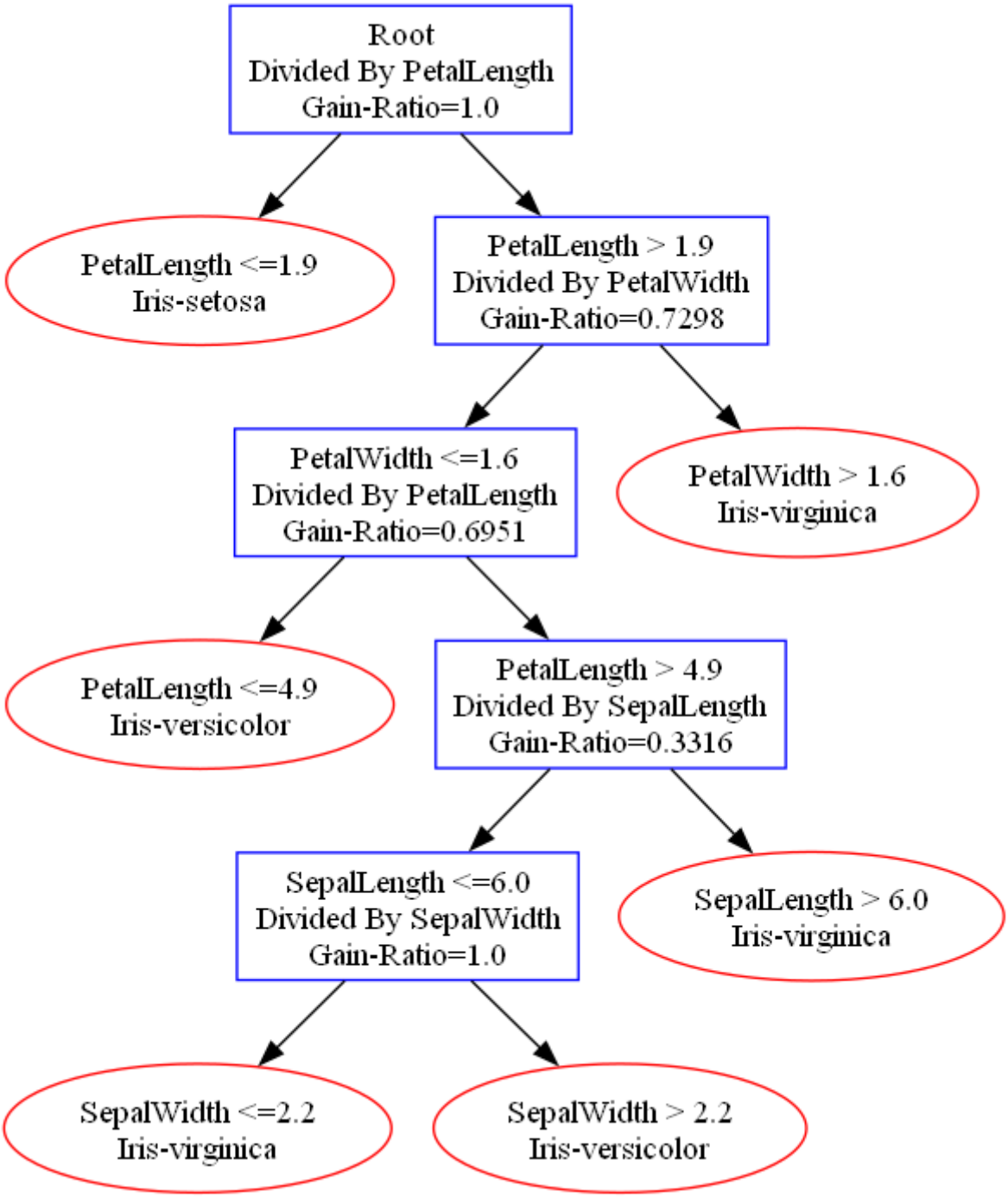
```
1  node = {
2      'type': 'leaf',
3      'class': self.result[np.bincount(y).argmax()]
4  }
```

6.3 后剪枝后结果对比

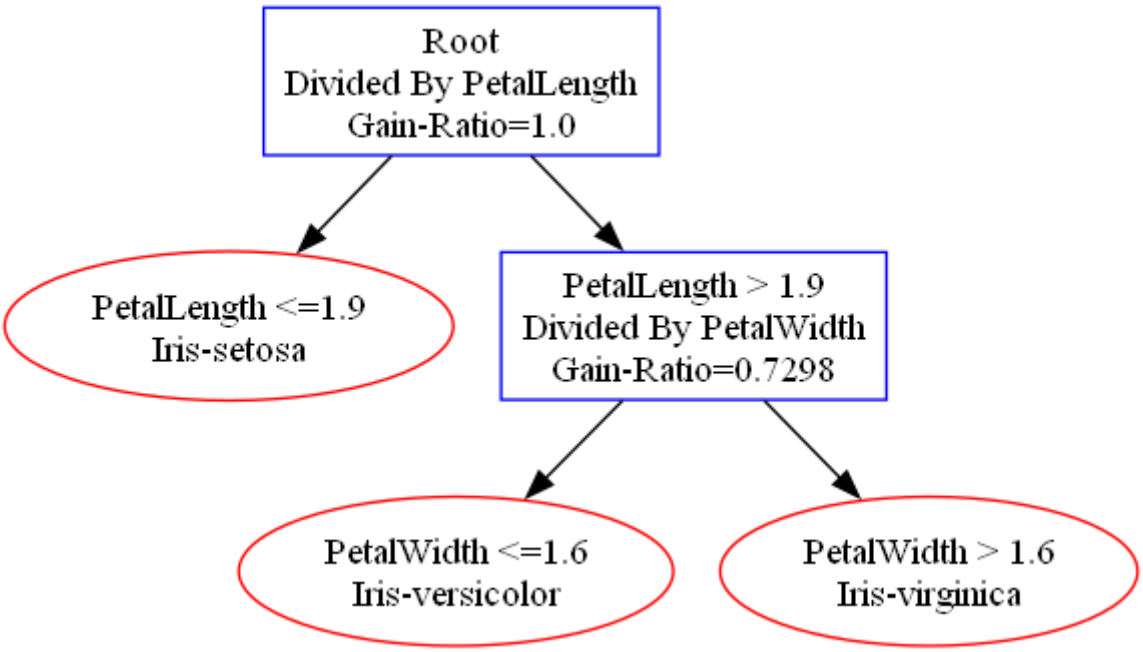
首先对两者同时对准确率进行测试：两者在验证集上的准确率和测试集上的准确率基本保持一致

但是两者的树状结构有着较大的差异，模型的整体复杂度大幅度降低，泛化能力会更好。

后剪枝前：



后剪枝后：



7.CART决策树的实现与剪枝优化

7.1 CART决策树的构建

CART决策树使用基尼指数(Gini index)来选择划分属性，数据集 $\mathbf{D}$ 的纯度可以用基尼值来衡量

$$Gini(\mathbf{D}) = \sum_{k=1}^{|y|}\sum_{k'\neq k} p_k p_{k'} = 1 - \sum_{k=1}^{|y|} p_k^2$$

直观来说，$\mathbf{D}$ 反应了从数据集 $\mathbf{D}$ 中随机抽取两个样本，其类别标记不一致的概率，因此$Gini(\mathbf{D})$越小，数据集$\mathbf{D}$ 的纯度越高，属性a 的基尼指数定义为：

$$Gini\_index(\mathbf{D};a) = \sum_{v=1}^{V}\frac{|\mathbf{D_v}|}{|\mathbf{D}|}Gini(\mathbf{D}_v)$$

于是从候选集中选择那个使得划分后基尼指数最小的属性作为最优划分属性，即

$$a_* = \underset{a\in A}{\mathbf{argmin}} Gini\_index(\mathbf{D};a)$$

CART模型的关键思想是使用二元划分（每次划分只产生两个子节点）来构建树结构。因此CART树一定是一颗二叉树（C4.5在离散值时不一定是二叉树）

7.2 代码实现

```
class CART:

    @staticmethod
    def splitForData(data, test_size, target, random_state=42):
        np.random.seed(random_state)
        data_shuffled = data.sample(frac=1).reset_index(drop=True)
        train_size = int((1 - test_size) * len(data))
        train_data = data_shuffled[:train_size]
        test_data = data_shuffled[train_size:]
        X_train = train_data.drop(target, axis=1)
        y_train = train_data[target]
        X_test = test_data.drop(target, axis=1)
        y_test = test_data[target]
```

```
        return X_train, y_train, X_test, y_test

    def __init__(self, max_depth=None, min_samples_split=2, max_gini=1):
        self.tree = None
        self.result = None

        # 预剪枝
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.max_gini = max_gini

    def train_val_split(self, X, y, test_size=0.2, random_state=None):
        # 设置随机种子
        if random_state:
            np.random.seed(random_state)

        # 打乱索引
        shuffled_indices = np.random.permutation(len(X))
        test_set_size = int(len(X) * test_size)
        test_indices = shuffled_indices[:test_set_size]
        train_indices = shuffled_indices[test_set_size:]

        return X.iloc[train_indices], X.iloc[test_indices],
y.iloc[train_indices], y.iloc[test_indices]

    def fit(self, X, y, isPruned=False, test_size=0.2, random_state=42):
        self.features = X.columns.tolist()
        # 创建从类别标签到索引的映射，并保存原始标签至self.result
        unique_labels = pd.unique(y)  # 获取y中的唯一值
        self.result = list(unique_labels)  # 保存标签列表

        # 转换y为索引值
        label_to_index = {label: index for index, label in
enumerate(unique_labels)}  # 创建从标签到索引的映射
        y_indexed = y.map(label_to_index)  # 将所有y值替换为对应的索引

        X_train, X_val, y_train, y_val = self.train_val_split(X, y_indexed,
test_size, random_state)

        # 构建决策树
        self.tree = self.build_tree(X_train, y_train, 1)
        if isPruned:
            self.tree = self.prune_tree(self.tree, X_val, y_val)
        return self.evaluate_accuracy(self.tree, X_val, y_val)

    def build_tree(self, X, y, depth):
        if len(np.unique(y)) == 1:
            return {'type': 'leaf', 'class': self.result[np.unique(y)[0]]}

        if self.max_depth and depth > self.max_depth:
            if len(y) == 0:
                return {'type': 'leaf', 'class': self.result[0]}
            return {'type': 'leaf', 'class':
self.result[np.bincount(y).argmax()]}

        # 预剪枝
        if len(y) < self.min_samples_split:
            if len(y) == 0:
```

```
                return {'type': 'leaf', 'class': self.result[0]}
            return {'type': 'leaf', 'class':
self.result[np.bincount(y).argmax()]}

        best_feature, best_threshold, best_gini = self.best_split(X, y)

        # 预剪枝
        # if best_gini > self.max_gini:
        #     return {'type': 'leaf', 'class':
self.result[np.bincount(y).argmax()]}

        left_indices = X[best_feature] <= best_threshold
        left_subtree = self.build_tree(X[left_indices], y[left_indices],
depth + 1)
        right_indices = X[best_feature] > best_threshold
        right_subtree = self.build_tree(X[right_indices], y[right_indices],
depth + 1)
        return {"feature": best_feature,
                "threshold": best_threshold,
                'gini': best_gini,
                'type': 'inner',
                "left": left_subtree,
                "right": right_subtree,
                "most_label": self.result[np.bincount(y).argmax()]}

    def best_split(self, X, y):
        best_gini = 1.0
        best_feature, best_threshold = None, None
        num_samples, num_features = X.shape
        for feature in X.columns:
            thresholds = np.unique(X[feature])
            for threshold in thresholds:
                gini = self.gini_index(X[feature], y, threshold)
                if gini < best_gini:
                    best_gini = gini
                    best_feature = feature
                    best_threshold = threshold
        return best_feature, best_threshold, best_gini

    def gini_index(self, X_feature, y, threshold):
        left_mask = X_feature <= threshold
        right_mask = X_feature > threshold
        left_gini = self.gini(y[left_mask])
        right_gini = self.gini(y[right_mask])
        p_left = float(np.sum(left_mask)) / len(y)
        p_right = float(np.sum(right_mask)) / len(y)
        return p_left * left_gini + p_right * right_gini

    def gini(self, labels):
        if labels.size == 0:
            return 0
        class_probs = np.array([np.sum(labels == c) for c in
np.unique(labels)]) / len(labels)
        return 1 - np.sum(class_probs ** 2)

    def predict(self, X):
        predictions = []
        for index, sample in X.iterrows():
```

```
            node = self.tree
            while isinstance(node, dict):
                if node['type'] == 'leaf':
                    break
                if sample[node['feature']] <= node['threshold']:
                    node = node['left']
                else:
                    node = node['right']
            predictions.append(node['class'])
        return predictions
```

整体构建部分与上述两颗决策树区别不大，仅在最优特征选取上进行了修改，采用了基尼指数的方式。在节点字典设计上采用了和C45Inspire的方式一致的设计，预测部分对于不存在边界的处理仍旧填充第一类。

内部节点：

```
node = {
    "feature": best_feature,
    "threshold": best_threshold,
    'gini': best_gini,
    'type': 'inner',
    "left": left_subtree,
    "right": right_subtree,
    # 为后剪枝准备
    "most_label": self.result[np.bincount(y).argmax()]
}
```

叶子节点：

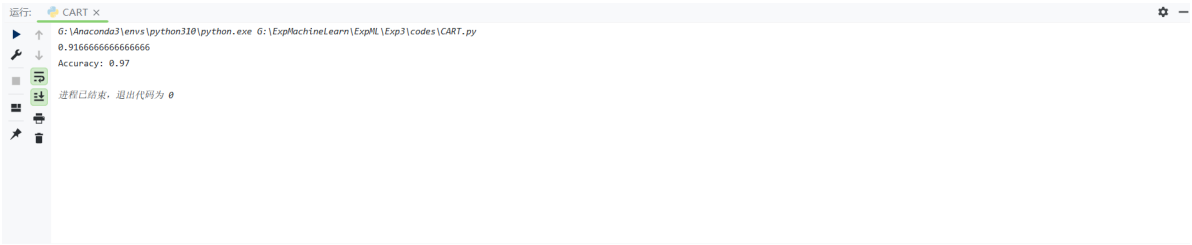
```
node = {
    'type': 'leaf',
    'class': self.result[np.bincount(y).argmax()]
}
```

7.3 测试集预测结果

编写主函数并且执行并获得预测结果

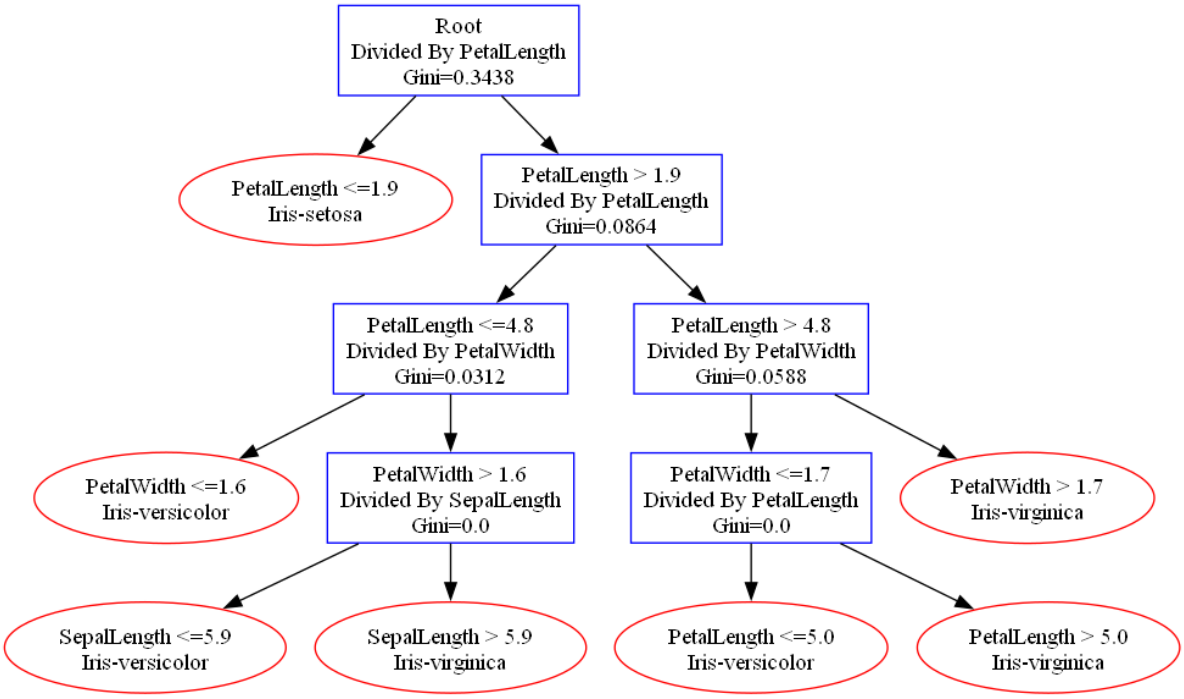
```
if __name__ == '__main__':
    df = pd.read_csv('../data/iris.csv')
    X_train, y_train, X_test, y_test = CART.splitForData(df, 0.2, 'Species',
random_state=227)

    tree = CART(max_depth=10)
    val_accuracy = tree.fit(X_train, y_train)
    print(val_accuracy)
    predictions = tree.predict(X_test)
    # print(predictions)
    accuracy = np.mean([predictions[i] == y_test.iloc[i] for i in
range(len(y_test))])
    print(f"Accuracy: {accuracy:.2f}")
```

执行后得到准确率为97%



7.4 CART可视化展示（三个预测过程展示）

利用`graphviz`建树的过程就不再赘述，建树后渲染后得到下述CART树可视化

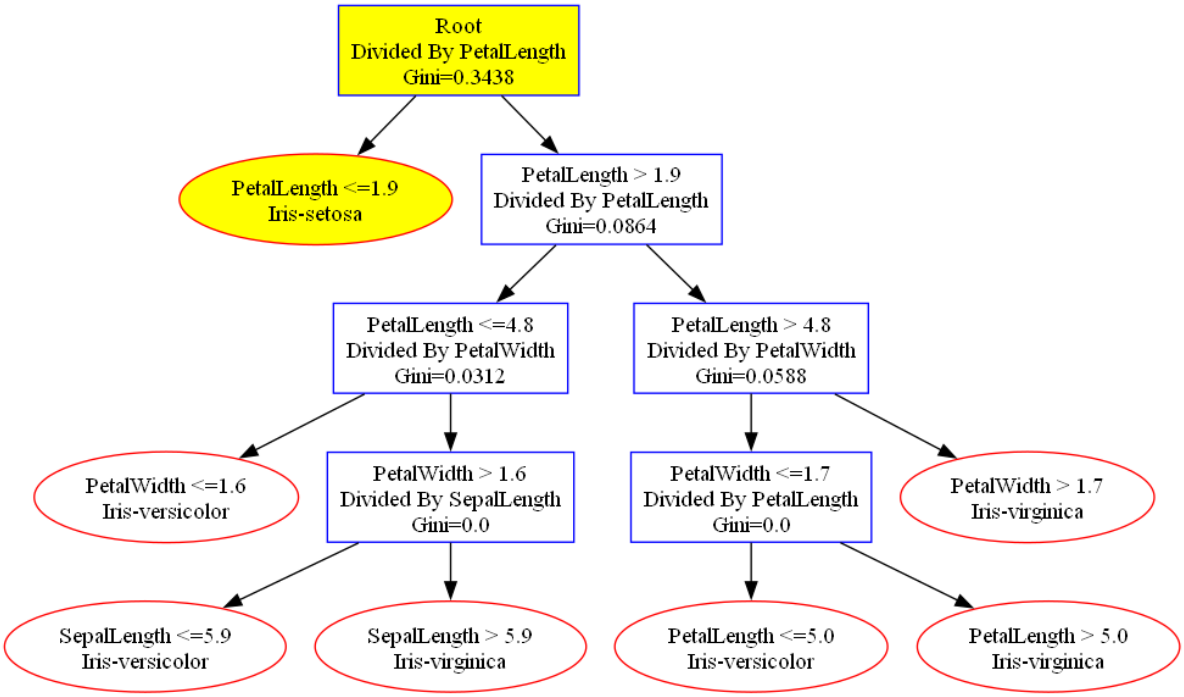


数据一：数据集中index=10的元素

数据：

```
SepalLength          5.4
SepalWidth           3.7
PetalLength          1.5
PetalWidth           0.2
```

搜索过程

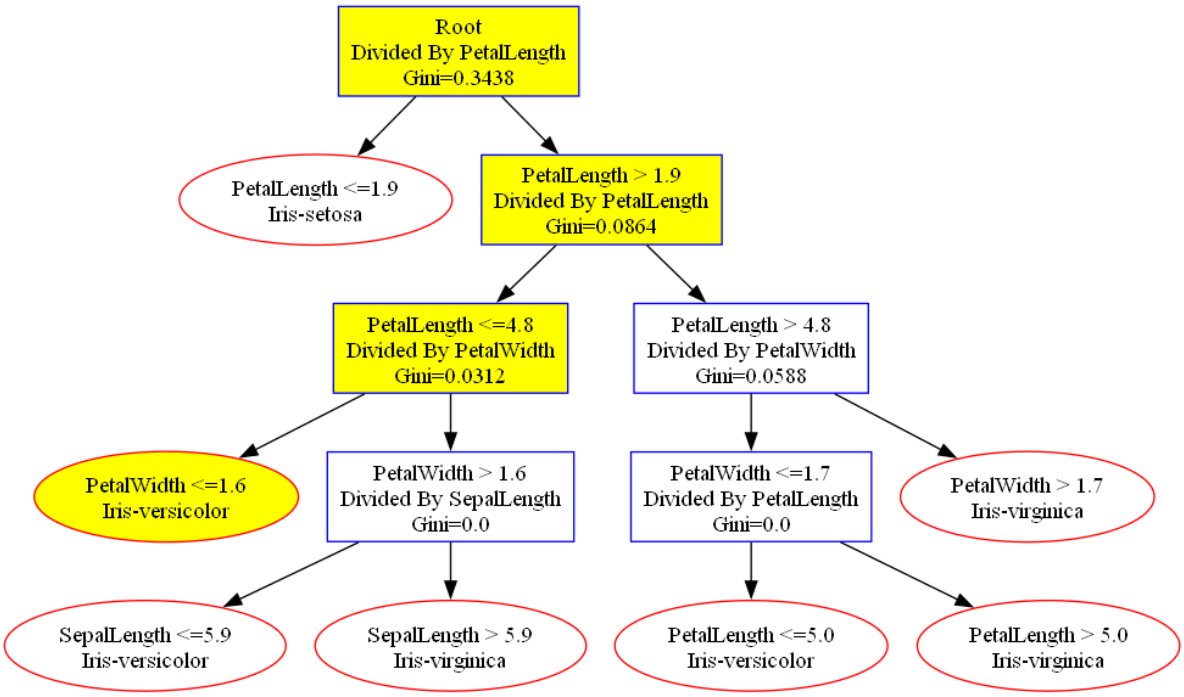


数据二：数据集中index=88的元素

数据：

```
SepalLength             5.6
SepalWidth              3.0
PetalLength             4.1
PetalWidth              1.3
```

搜索过程

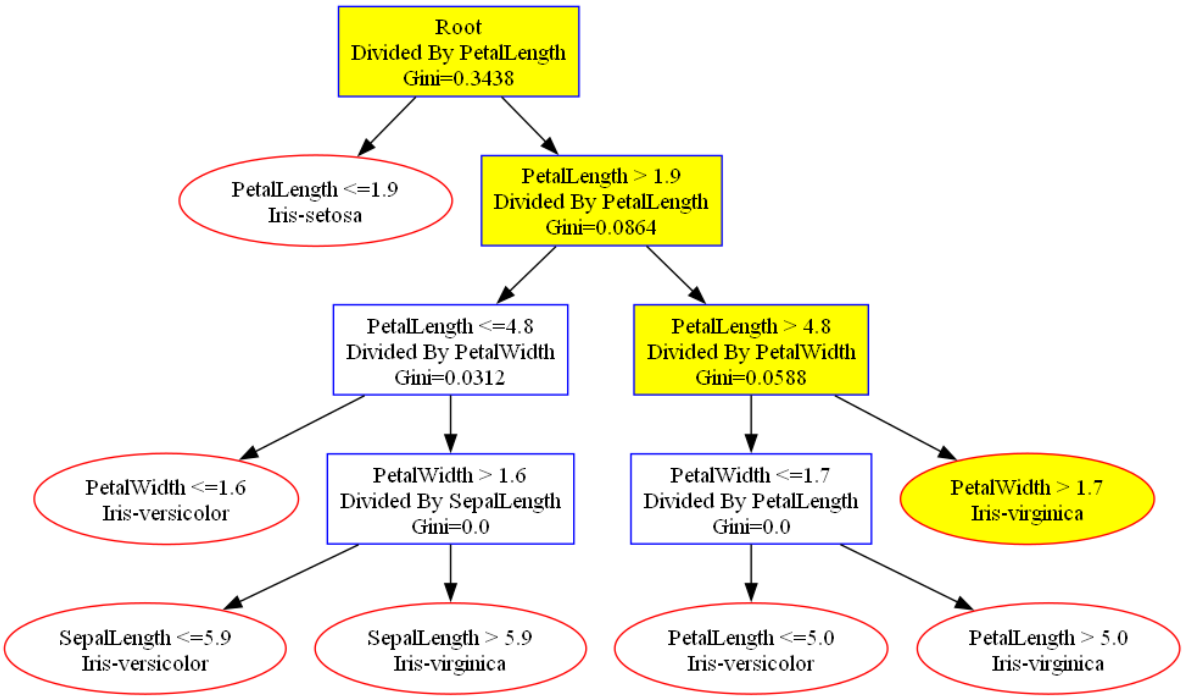


数据三：数据集中index=120的元素

数据：

```
SepalLength             6.9
SepalWidth              3.2
PetalLength             5.7
PetalWidth              2.3
```

搜索过程



7.5 CART后剪枝（CCP）

在构建CART树的过程种，显然我已经对其进行了预剪枝，在此处着重阐述基于代价复杂度的后剪枝（Cost-Complexity Pruning, CCP）。

CCP后剪枝策略的基本步骤

1. **完全生长树**： 首先，构建一棵完全生长（未剪枝）的决策树，即树的每一个叶节点都只包含同一类的样本或达到最小分割样本数。
2. **计算代价复杂度**： 代价复杂度（Cost-Complexity）是模型误差和树的复杂度的加权和。对于每个子树T_t，其代价复杂度定义为：

   $$R_\alpha(T_t) = R(T_t) + \alpha|T_t|$$

   其中：
   - $R(T_t)$ 是子树 T_t 的误差（例如，分类误差或回归误差）。
   - $|T_t|$ 是子树 T_t 的叶节点数量，表示树的复杂度。
   - α 是一个非负参数，用于权衡误差和复杂度。
3. **选择最佳剪枝点**： 通过选择不同的 α 值，计算不同子树的代价复杂度。对每个内部节点，计算将该节点及其子树替换为叶节点后的代价复杂度。如果剪枝后的代价复杂度小于不剪枝的代价复杂度，则进行剪枝。
4. **生成剪枝路径**： 通过逐步增加 α 值，生成一系列从完全生长树到仅有根节点的树的序列。这个序列被称为剪枝路径，每个路径上的树对应一个特定的 α 值。
5. **选择最优子树**： 使用交叉验证选择剪枝路径上最优的子树。对于每一个 α 值对应的子树，计算其在验证集上的性能，选择在验证集上表现最好的那棵树作为最终模型。

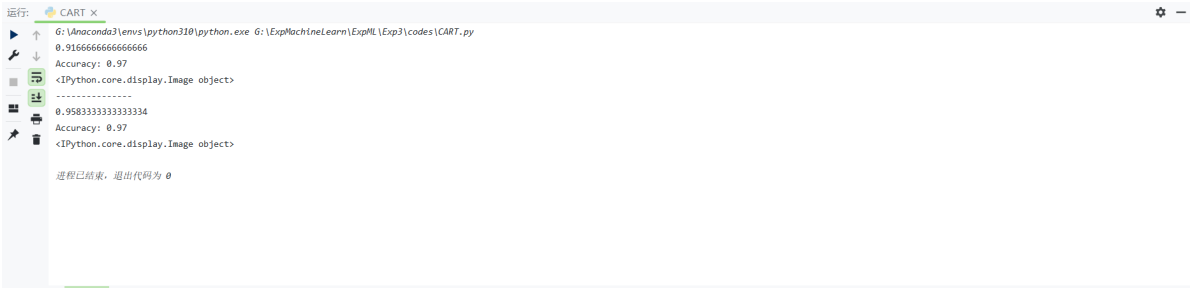
在代码中引入这部分的实现，如下：

```
    def cost_complexity_pruning(self, tree, X_val, y_val):
        def tree_cost(node):
            if node['type'] == 'leaf':
                return 0
            else:
                return 1 + tree_cost(node['left']) +
tree_cost(node['right'])

        def subtree_error(node, X, y):
            if node['type'] == 'leaf':
                pred_class = node['class']
                return np.sum([pred_class != self.result[y.iloc[i]] for i in
range(len(y))])
            else:
                feature = node['feature']
                threshold = node['threshold']
                left_indices = X[feature] <= threshold
                right_indices = X[feature] > threshold
                left_error = subtree_error(node['left'], X[left_indices],
y[left_indices])
                right_error = subtree_error(node['right'], X[right_indices],
y[right_indices])
                return left_error + right_error

        def prune(node, X, y, alpha):
            if node['type'] == 'leaf':
                return node, subtree_error(node, X, y)
            else:
                feature = node['feature']
                threshold = node['threshold']
                left_indices = X[feature] <= threshold
                right_indices = X[feature] > threshold

                node['left'], left_error = prune(node['left'],
X[left_indices], y[left_indices], alpha)
                node['right'], right_error = prune(node['right'],
X[right_indices], y[right_indices], alpha)

                leaf_error = subtree_error({'type': 'leaf', 'class':
node['most_label']}, X, y)
                total_error = left_error + right_error
                if leaf_error + alpha * tree_cost(node) < total_error +
alpha * tree_cost(node):
                    return {'type': 'leaf', 'class':
self.result[np.bincount(y).argmax()]}, leaf_error
                else:
                    return node, total_error

        alphas = np.linspace(0, 0.1, 101)
        best_tree = tree
        best_acc = self.evaluate_accuracy(tree, X_val, y_val)

        for alpha in alphas:
            pruned_tree, _ = prune(tree, X_val, y_val, alpha)
            acc = self.evaluate_accuracy(pruned_tree, X_val, y_val)
```

```
            if acc >= best_acc:
                best_acc = acc
                best_tree = pruned_tree

        return best_tree

    def prune_tree(self, tree, X_val, y_val):
        return self.cost_complexity_pruning(tree, X_val, y_val)
```

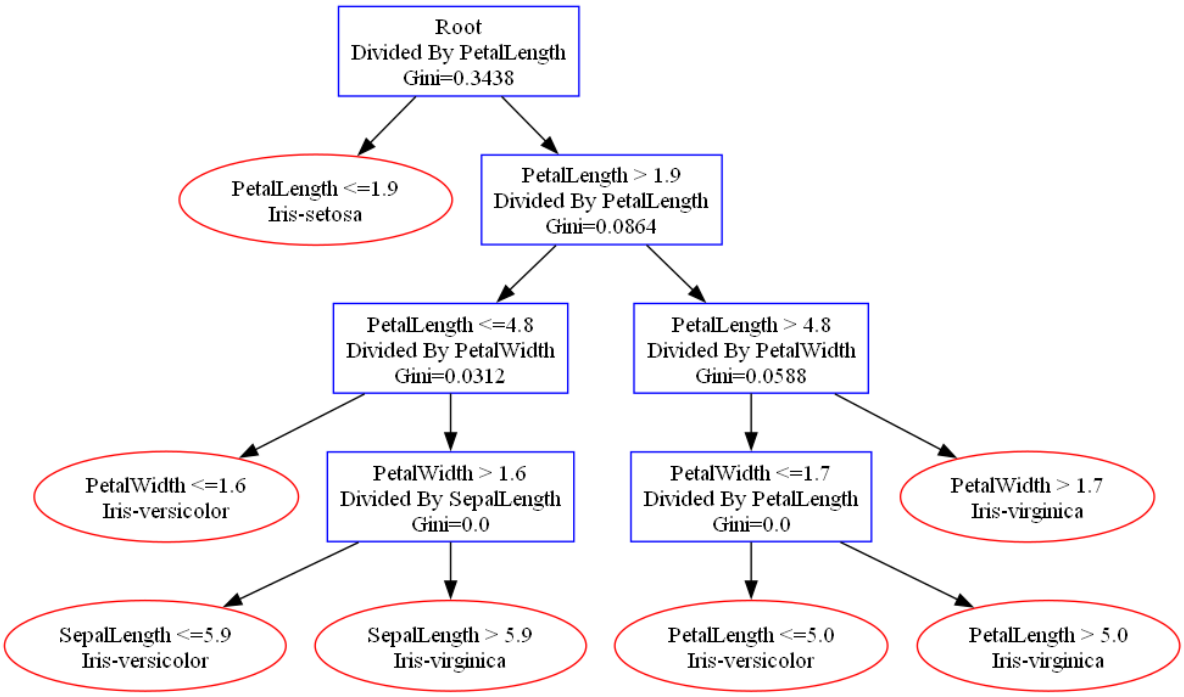
7.6 CART后剪枝结果对比

剪枝前后验证集和测试集准确率上对比如下：可以发现验证集上的准确率有着较大提升，测试集准确率保持不变（由于测试集数据量太少导致）

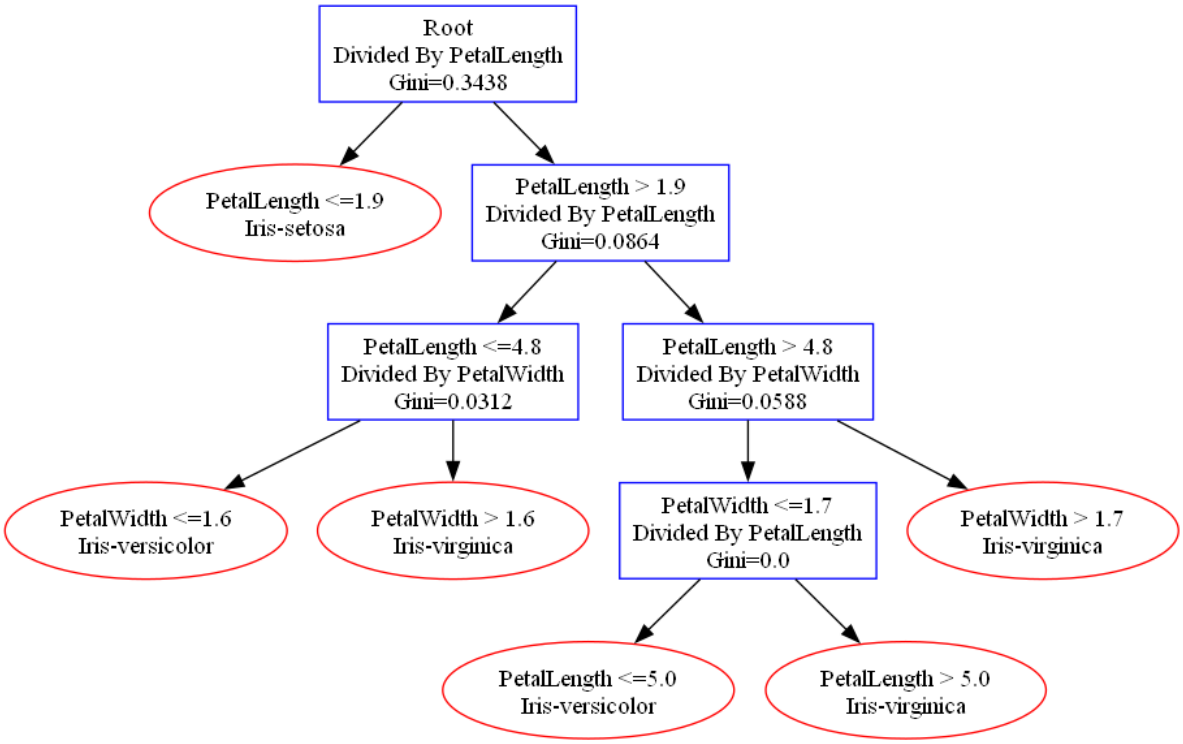


树的结构对比如下：

剪枝前：



剪枝后：



8. C4.5剪枝和CART剪枝的异同

C4.5和CART是两种广泛使用的决策树算法，它们在剪枝策略上有一些显著的相似点和不同点。

8.1 相同点

1. **目的相同**：两者的剪枝方法都是为了减少过拟合，提高决策树在未见数据上的泛化能力。
2. **后剪枝策略**：C4.5和CART的剪枝都是后剪枝，即在生成一棵完整的树之后进行剪枝操作。
3. **基于验证集**：两者都可以使用验证集上的性能作为剪枝的依据。

8.2 不同点

1. **剪枝准则**：

   C4.5：使用错误率剪枝（Error-based Pruning），具体来说，它计算剪枝前后的错误率差异，通过统计检验来决定是否剪枝。具体步骤包括计算子树和父节点的错误率，并使用校正错误率（例如使用拉普拉斯校正）来决定是否剪枝。

   ```
   计算当前子树的错误率 e1 和父节点的错误率 e2，如果 e1 > e2，进行剪枝。
   ```

   CART：使用代价复杂度剪枝（Cost-Complexity Pruning, CCP），它通过计算每个子树的代价复杂度来决定是否剪枝。代价复杂度包含误差和模型复杂度两部分，即：

   $$R_\alpha(T_t) = R(T_t) + \alpha|T_t|$$

   其中$R(T_t)$是子树的误差，$|T_t|$是子树的叶节点数量，α是权衡参数。CART通过选择不同的α值生成一系列子树，然后通过交叉验证选择最优子树。

2. **计算复杂度**：

   C4.5：由于使用错误率剪枝，相对来说计算复杂度较低，只需要比较错误率和进行简单的统计检验。

   CART：CCP需要计算代价复杂度，这涉及到对树结构的更复杂的遍历和计算，计算复杂度较高。

3. **剪枝过程**：

   C4.5：在剪枝过程中，C4.5倾向于将较小的子树合并成一个叶节点，如果合并后的叶节点在验证集上的表现优于原来的子树。

   CART：CART的CCP通过逐步增加 α 值来生成一系列树，然后选择最优的那棵。这种方式更系统化，可以更好地控制模型的复杂度。

4. **结果解释**：

   C4.5：剪枝后的树仍然保留原始的决策路径，只是部分路径被截断。结果树依然基于信息增益比来做决策。

   CART：剪枝后的树在结构上可能有显著变化，尤其是在选择最优子树的过程中，结果树的决策路径和原始树可能有较大不同。

代码结构

```
/Exp3/
|------- code/
|        |------- C45.py 带预剪枝的C4.5决策树
|        |------- C45Inspire.py 带后剪枝的C4.5决策树
|        |------- CART.py 带前后剪枝的CART决策树
|        |------- ID3Continous.py 连续值ID3决策树
|        |------- ID3Discrete.py 离散值ID3决策树
|        |------- SplitData.py 数据分割代码
|
|-------- data/
|        |------- iris.csv 数据集
|
|-------- pic/
|        |------- C45  C4.5有关的树状图像和graphviz文件
|        |------- CART CART有关树状图像和graphviz文件
|        |------- ID3  ID3有关树状图像和graphviz文件
|        |------- submit 部分实现截图
|
|-------- 实验报告.md 实验报告Markdown
|
|-------- 实验报告.pdf 实验报告pdf
```

心得体会

通过这次实验，我不仅加深了对决策树算法的理论理解，也掌握了如何在实际操作中实现这些算法。特别是在实现C4.5和CART的过程中，我遇到了一些细节上的挑战，比如如何设置剪枝条件，如何平衡模型复杂度和泛化能力，这些都需要理论知识与实际操作的紧密结合。

实验中，我体会到了数据预处理对模型性能的重要性。无论是处理缺失值，还是进行数据标准化，这些步骤都会对最终的模型效果产生显著影响。在实际应用中，良好的数据预处理不仅能提升模型的准确性，还能减少过拟合的风险。

通过对比C4.5和CART的剪枝策略，我了解到不同的剪枝方法在效果上有显著差异。C4.5采用错误率剪枝，计算复杂度相对较低，但在某些情况下可能不足以处理复杂的树结构。而CART的代价复杂度剪枝尽管计算复杂度较高，但在控制模型复杂度和提高泛化能力方面表现更佳。这让我意识到，在实际应用中，选择合适的剪枝策略是至关重要的。

在编写实验代码时，我深刻体会到代码结构清晰、模块化的重要性。将不同算法的实现、数据处理和结果展示分模块编写，不仅提高了代码的可读性和可维护性，也使得调试和优化变得更加方便。良好的代码结构是高效实验和研究的基础。

通过实验，我学会了如何有效地分析和展示实验结果。图表和可视化工具的使用，让我能够直观地观察到不同剪枝策略对树结构和模型性能的影响。清晰的结果展示不仅帮助我更好地理解实验现象，也有助于向他人传达实验成果。

尽管实验取得了一定的成果，但我也发现了一些不足之处。例如，在某些数据集上的剪枝效果不如预期，可能是由于剪枝条件设置不合理或数据集本身的特性导致的。在未来的实验中，我计划通过更多的数据集测试和参数调优，进一步提高模型的鲁棒性和适用性。

总体而言，本次实验让我对决策树算法有了更全面和深入的理解，也提高了我在数据科学领域的实践能力。通过不断的学习和实践，我相信自己能够在今后的研究中取得更好的成果。