Heart Disease Detection: A CART-Based Binary Classification Model

Name: Yixun Kang, David Ning, Liang Zhang

Team: UC Providence

Link to the github repo: https://github.com/ihiroo/DATA2060_Final_Project

In [3]:
```
# Dependencies
import numpy as np
from sklearn.datasets import load_iris
from graphviz import Digraph
from sklearn import tree
import matplotlib.pyplot as plt
import random
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split, KFold
```

In [4]:
```
# Set seed for python and numpy
np.random.seed(0)
random.seed(0)
```

Overview:

1 Overview

1.1 CART

Classification and Regression Trees (CART) is a supervised machine learning algorithm that constructs decision trees to make predictions. In this project, CART is employed for binary classification, where the target variable has only two possible outcomes. Also compared against a previous work which has multi-class target.

CART models are highly interpretable because the decision rules can be visualizes as a tree structure, containing decision nodes and leaf nodes. When building up the tree, CART can handle both continuous and categorical features without complex preprocessing. Moreover, CART is non-parametric as it does not assume any specific distribution for the data, and can handle non-linear relationships. Since our dataset contains 11 continuous features and 2 categorical features, and we are working with a relatively small dataset, CART is a suitable choice. Its ability to handle mixed data types seamlessly and its efficiency on smaller datasets make it an effective and practical algorithm for this task.

While CART offers many advantages, it also has notable limitations. One major drawback is its tendency to overfit the training data, especially when the tree grows too deep. This can lead to poor generalization on unseen data, making techniques like pruning essential. Additionally, CART is instable, as even small changes in the training data can result in significant variations in the tree structure. Finally, while CART is effective for many problems, its predictive performance may fall short compared to more advanced ensemble methods like Random Forest or Boosting, which can leverage multiple trees to produce more robust predictions.

1.2 Data

The dataset for this project is the Heart Disease dataset, sourced from UCI Machine Learning Repository [2]. It's target variable is binary, with a value of 0 indicating the absence of heart disease and a value of 1 indicating its presence. The dataset contains 13 features that capture a variety of patient characteristics and diagnostic indicators, including both continuous, ordinal and categorical variables. Since ordinal features are encoded as numeric values, they will be treated as continuous features in the modeling process. For features `sex` and `exang`, they are binary categorical but are encoded as numeric values as well.

This project will also use the Iris dataset, sourced from UCI Machine Learning Repository [1] to compare the implementation from scratch with the Scikit-learn model. The Iris dataset contains 4 continuous features. Its target variable is categorical, with 3 classes, representing the species of the iris flower.

1.3 Representation

The CART algorithm predicts a single outcome for binary classification by recursively partitioning the dataset using decision rules based on the input features. We require 4 inputs for the algorithm , including the training dataset S, feature subset A, and two values for the hyperparameters. When building the tree, if none of the stopping conditions are met, we first compute the Gini impurity for the current dataset, denoted as $\text{Gini}(S)$. Then for each feature $\mathbf{x}_i$ within the current feature subset A, we sorted the unique values of each feature $\mathbf{x}_i$ and generate a sequence of midpoints based on the sorted values. For each midpoint, or threshold θ, the current dataset is split into two subsets, S_1 and S_2. Subset S_1 contains data points where $x_i \leq \theta$, and subset S_2 contains data points where $x_i > \theta$, with x_i representing a value within the feature $\mathbf{x}_i$. Next, we calculate the Gini

impurity for both subsets, S_1 and S_2, as well as the weighted Gini impurity, denoted as $\text{Gini}(S_1, S_2, S)$. We measure the gain as the difference between the node Gini impurity $\text{Gini}(S)$ and the weighted Gini impurity and choose the feature $\mathbf{x}_j$ and threshold combination that will maximize this gain. Based on the chosen feature $\mathbf{x}_j$ and threshold θ, we split the current dataset into two subset, S_1 and S_2, where subset S_1 contains data points where $x_i \leq \theta$, and subset S_2 contains data points where $x_i > \theta$. Using the strategy, we will recursively build the tree until one of the stopping conditions is met. Since in the Heart Disease dataset, all features are encoded as numeric values, this algorithm is enough for handling 13 numeric features. However, if the ordinal and categorical features are not encoded yet, this algorithm will not work and further modifications to the algorithm are required.

We used 5 stopping conditions to terminate the recursion while building the tree. First, if all samples in the current subset S are labeled as 1, a leaf node labeled 1 will be returned. Second, if all samples are labeled as 0, a leaf node labeled 0 will be returned. These two conditions ensure that the recursion stops when the node becomes pure, meaning all samples in the subset belong to the same class, and further splitting is unnecessary. Third, if the feature subset A is empty, the recursion stops, and a leaf node will be returned with its value set to the majority label of the current subset S. This condition acts as a safeguard to prevent further splits when no features are left to evaluate, ensuring that the tree construction terminates gracefully even if the dataset cannot be split further in a meaningful way. Fourth, if the maximum tree depth is reached, the recursion terminates, and a leaf node will be returned with its value equal to the majority class label in S. The maximum tree depth is determined by the hyperparameter `max_depth` . Finally, if the size of the current subset $|S|$ is smaller than a predefined minimum number of samples (`min_samples_split`) required to split, a leaf node will be returned, with its value set to the majority class label in S . In the last three conditions, assigning the majority class label ensures that the algorithm no only provides a reasonable prediction even when further splitting is restricted by depth constraints, but also works for multi-class classification.

Algorithm 1 CART for Binary Classification

Inputs: (i) training data S; (ii) feature subset $A \subseteq [d]$; (iii) max_depth; (iv) min_samples_split

if all samples in S are labeled by 1 **then** return a leaf node labeled by 1
if all samples in S are labeled by 0 **then** return a leaf node labeled by 0
if $A = \emptyset$ **then** return a leaf whose value = majority of labels in S
if max_depth $= 0$ **then** return a leaf whose value = majority of labels in S
if $|S| <$ min_samples_split **then** return a leaf whose value = majority of labels in S
else
  Compute $\text{Gini}(S) \leftarrow 1 - \{[\Pr(y = 1|S)]^2 + [\Pr(y = 0|S)]^2\}$
  foreach feature i in A **do**
    Generate a sequence of thresholds θ based on the sorted unique values of $\mathbf{x}_i$
      foreach threshold θ in the sequence **do**
        $S_1 = \{(\mathbf{x}, y) \in S : x_i \leq \theta\}$
        $S_2 = \{(\mathbf{x}, y) \in S : x_i > \theta\}$
        $\text{Gini}(S_1, S_2, S) = \frac{|S_1|}{|S|} \cdot \text{Gini}(S_1) + \frac{|S_2|}{|S|} \cdot \text{Gini}(S_2)$
        $\text{Gain}(S_1, S_2, S) = \text{Gini}(S) - \text{Gini}(S_1, S_2, S)$
  Let $j = \text{argmax}_{i \in A} \text{Gain}(S_1, S_2, S)$
  $S_1 = \{(\mathbf{x}, y) \in S : x_j \leq \theta\}$
  $S_2 = \{(\mathbf{x}, y) \in S : x_j > \theta\}$
  Let T_1 be the tree returned by $\text{CART}(S_1, A, \text{max\_depth} - 1, \text{min\_samples\_split})$
  Let T_2 be the tree returned by $\text{CART}(S_2, A, \text{max\_depth} - 1, \text{min\_samples\_split})$
Return the tree

1.4 Loss

CART has 3 ways to calculate the loss, including the Gini impurity, entropy and misclassification error. This project uses Gini impurity since it's the default criterion in Scikit-learn's `DecisionTreeClassifier`. We will calculate the loss recursively when building up the tree. The parent Gini impurity, or the node Gini impurity is calculated as

$$\text{Gini}(S) = 1 - \sum_{k=1}^{K} p_k^2,$$

where K is the number of unique classes in the entire dataset, and p_k is the proportion of samples in the subset that belong to class k. In a binary classification problem,

$$\text{Gini}(S) = 1 - \sum_{k=1}^{2} [P(y = k|S)]^2 = 1 - (p_1^2 + p_2^2),$$

The weighted Gini impurity is calculated as

$$\text{Gini}(S_1, S_2, S) = \frac{|S_1|}{|S|} \cdot \text{Gini}(S_1) + \frac{|S_2|}{|S|} \cdot \text{Gini}(S_2),$$

where $\text{Gini}(S_1)$ and $\text{Gini}(S_2)$ are the Gini impurity of the two splitted subsets S_1 and S_2, and we use $|\cdot|$ to measure the cardinality, which is the number of samples within one dataset.

Then, the gain from split is measured as the difference between the parent Gini impurity and the weighted Gini impurity. We then have

$$\text{Gain}(S_1, S_2, S) = \text{Gini}(S) - \text{Gini}(S_1, S_2, S).$$

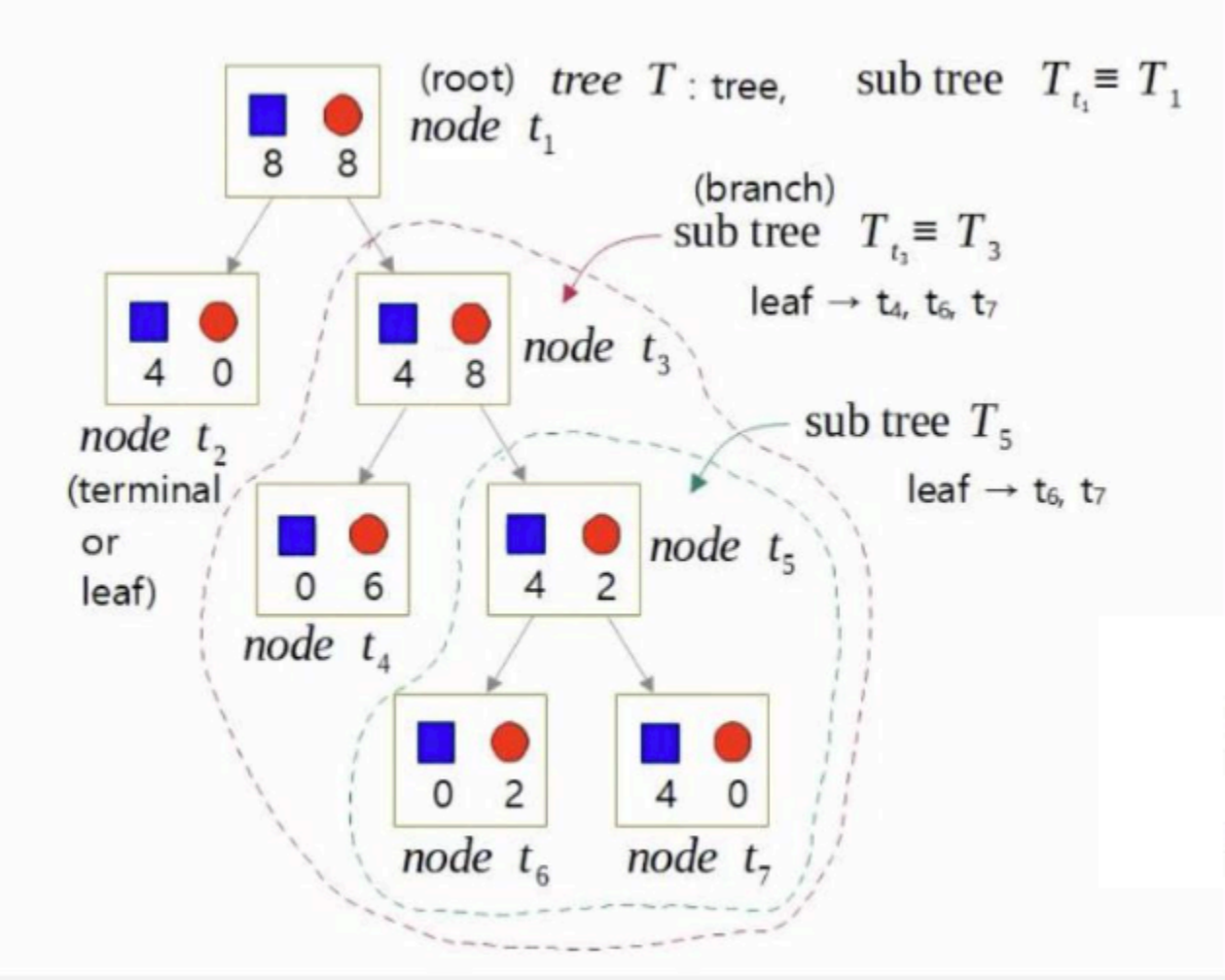
1.5 Optimizer: Pruning

Pruning is used to optimize the DecisionTree classifier. In Scikit-learn, the DecisionTree classifier uses a parameter called cost-complexity pruning (ccp_alpha) to control the trade-off between the complexity of the tree and the loss. The reason why we need this is that pruning helps improve the generalization by cutting off unnecessary branches, thus preventing overfitting. During the pruning process, the algorithm will compute a cost-complexity measure for all the subtrees and add a penalty equal to ccp_alpha times the number of leaves in the subtrees.

The pruning process starts from top to bottom, which means that it goes from the root and recursively selects each subtree and their children subtrees to perform this optimization. As shown in the graph below[6]. In the end, it should simplify the tree and reduce overfitting. The global cost function is written as $R_\alpha(T) = R(T) + \alpha * |T|$, T is the subtree, $R(t)$ is the risk of the tree, which can be chosen from overall/Gini/entropy/etc. $R(T_t)$ is the risk of a certain node that is going to be pruned and $|T|$ is the number of terminal nodes. The local pruning rule states that if $\alpha > \frac{R(t)-R(T_t)}{|T_t|-1}$, we can change this equation to this: prune

T_t if $(|T_t| - 1) * \alpha > R(t) - R(T_t)$ We iterate through different values of α to find the optimal value that best improves the performance of the classifier.

We also use cross-validation, specifically 5-fold cross-validation, for evaluating model performance by splitting the training data into 5 subsets. Initially, we split the entire dataset into 80% train and 20% test, then for the training dataset, we apply the 5-fold cross-validation to ensure robustness. In each iteration of the cross-validation process, it chooses the i-th fold as validation and the rest as training. The model is trained and validated iteratively across all folds to achieve reliable performance measures.



Model Section

In [5]:
```
class Node:
    '''
    Constructor for the Node class
    '''
    def __init__(self, left=None, right=None, label=None, feature=None, threshold=None, parent_gini=None, nc
        self.left = left # to a left node
        self.right = right # to a right node
        self.label = label
        self.feature = feature
        self.threshold = threshold
        self.parent_gini = parent_gini
        self.node_gini = node_gini
        self.num_samples = num_samples # data points in this node
        self.class_counts = class_counts

    def is_leaf(self):
        '''
        check if the node is a leaf node
        '''
        return self.label is not None
```

In [87]:
```
class CART:
    '''
    Decision Tree classifier by UC Providence
    '''
    def __init__(self, max_depth=None, min_samples_split=2, ccp_alpha=0.01, random_state=0):
        if max_depth is None:
            self.max_depth = 20
        else:
            self.max_depth = max_depth # set the max depth
        self.min_samples_split = min_samples_split
        self.tree = None
        self.ccp_alpha = ccp_alpha # for pruning
        self.random_state = random_state
        if random_state is not None: # make random state deterministic
            np.random.seed(random_state)
            random.seed(random_state)

    def fit(self, data):
        '''
```

```
        build the tree based on the data
        '''
        self.tree = self._build_tree(data, depth=0)

    def prune(self, data, ccp_alpha=0):
        '''
        prune the tree based on the ccp_alpha
        '''
        self._prune_tree(data, self.tree, ccp_alpha)

    def predict(self, data):
        '''
        Helper function to predict the data
        '''
        X = data[:, :-1] # get the features
        return np.array([self._predict_row(self.tree, row) for row in X])

    def loss(self, data):
        '''
        Helper function to calculate the loss
        '''
        preds = self.predict(data)
        true_labels = data[:, -1] # last column is the label
        return np.sum(preds != true_labels) / len(true_labels)

    def accuracy(self, data):
        '''
        Helper function to calculate the accuracy
        '''
        return 1 - self.loss(data)

    def _gini_for_node(self, data):
        '''
        Get the gini index for a node
        params data: the data in the node
        return: the gini index
        '''
        labels = data[:, -1] # get the last column which is the label
        _, counts = np.unique(labels, return_counts=True)
        probs = counts / len(labels)
        parent_gini = 1 - np.sum(probs ** 2) # calculate the gini index
        return parent_gini

    def _gini_for_split(self, data, left, right):
        '''
```

```
        Get the gini index for a split
        params data: the data in the node
        params left: the left split
        params right: the right split
        return: the gini index
        '''
        # calc the total size
        total_size = len(data)
        left_size = len(left)
        right_size = len(right)
        # calc the gini index for the left and right
        left_gini = self._gini_for_node(left)
        right_gini = self._gini_for_node(right)
        # calc the weighted gini index
        weighted_gini = (left_size / total_size) * left_gini + (right_size / total_size) * right_gini
        return weighted_gini

    def _split(self, data, feature_index, threshold):
        '''
        Split the data based on the feature and threshold
        params data: the data
        params feature_index: the feature to split on
        params threshold: the threshold to split on
        return: the left and right split
        '''
        left = data[data[:, feature_index] <= threshold]
        right = data[data[:, feature_index] > threshold]
        return left, right

    def _find_best_split(self, data):
        '''
        Find the best split for the data, traverse through each column and each average value of the values
        params data: the dataset
        return: the best gain and the best split
        '''
        best_gain = float("-inf")
        best_split = None
        best_split_list = [] # for ties
        parent_gini = self._gini_for_node(data) # calc the gini index for the parent node
        n_features = data.shape[1] - 1
        for feature in range(n_features): # traverse through each feature
            unique_values = np.unique(data[:, feature])
            sorted_values = np.sort(unique_values)
            thresholds = (sorted_values[1:] + sorted_values[:-1]) / 2 # get the average of the values
```

```
            if len(thresholds) > 2:
                # Continuous or ordinal features
                for threshold in thresholds:
                    left, right = self._split(data, feature, threshold)
                    if len(left) == 0 or len(right) == 0:
                        continue # skip if the split is empty
                    weighted_gini = self._gini_for_split(data, left, right) # calc the weighted gini index
                    gain = parent_gini - weighted_gini
                    if gain > best_gain: # if the gain is better than the best gain
                        best_gain = gain
                        best_split_list = [{
                            "feature": feature,
                            "threshold": threshold,
                            "gini_for_split": weighted_gini,
                            "parent_gini": parent_gini,
                            "gain": gain,
                            "left": left,
                            "right": right,
                            "type": "continuous"
                        }]
                    elif np.isclose(gain, best_gain):
                        best_gain = gain
                        best_split_list.append({
                            "feature": feature,
                            "threshold": threshold,
                            "gini_for_split": weighted_gini,
                            "parent_gini": parent_gini,
                            "gain": gain,
                            "left": left,
                            "right": right,
                            "type": "continuous"
                        }) # if tied, then append to the list
            else:
                # Only one threshold for binary features
                for threshold in thresholds:
                    left, right = self._split(data, feature, threshold)
                    if len(left) == 0 or len(right) == 0:
                        continue
                    weighted_gini = self._gini_for_split(data, left, right)
                    gain = parent_gini - weighted_gini
                    if gain > best_gain:
                        best_gain = gain
                        # same for binary features but with different type
                        best_split_list = [{
                            "feature": feature,
```

```
                            "threshold": threshold,
                            "gini_for_split": weighted_gini,
                            "parent_gini": parent_gini,
                            "gain": gain,
                            "left": left,
                            "right": right,
                            "type": "binary"
                        }]
                    elif np.isclose(gain, best_gain):
                        best_gain = gain
                        best_split_list.append({
                            "feature": feature,
                            "threshold": threshold,
                            "gini_for_split": weighted_gini,
                            "parent_gini": parent_gini,
                            "gain": gain,
                            "left": left,
                            "right": right,
                            "type": "binary"
                        })
        if len(best_split_list) > 1:
            # if the best split list has more than one best split then we sort it by feature
            #print("Multiple best splits found")
            #x = sorted(best_split_list, key=lambda x: x["feature"])
            # for i in x:
            #     print(i["feature"], i["threshold"], i["gini_for_split"], i["parent_gini"], i["gain"])
            best_split = sorted(best_split_list, key=lambda x: x["feature"])[-1]
        else: # else we just get the first best split
            best_split = best_split_list[0]
        return best_gain, best_split

    def _majority_class(self, data):
        '''
        Get the majority class in the data
        '''
        labels = data[:, -1] # get the labels
        unique_labels, counts = np.unique(labels, return_counts=True)
        return unique_labels[np.argmax(counts)]

    def _build_tree(self, data, depth=0):
        '''
        Build the tree recursively
        params data: the data
        params depth: the depth of the tree
        return: the node and its attributes
```

```
        '''
        labels = data[:, -1] # label is the last column
        num_samples = len(labels)
        parent_gini = self._gini_for_node(data)

        # Stopping conditions
        # Having a pure node
        if len(np.unique(labels)) == 1:
            return Node(label=labels[0], parent_gini=parent_gini, num_samples=num_samples)
        # Max depth reached
        if self.max_depth is not None and depth >= self.max_depth:
            return Node(label=self._majority_class(data), parent_gini=parent_gini, num_samples=num_samples)
        # Minimum samples split reached
        if num_samples < self.min_samples_split:
            return Node(label=self._majority_class(data), parent_gini=parent_gini, num_samples=num_samples)
        # No split found
        best_gain, best_split = self._find_best_split(data)
        if best_gain == 0:
            return Node(label=self._majority_class(data), parent_gini=parent_gini, num_samples=num_samples)

        # Put left and right data into the tree
        if best_split["type"] == "binary":
            remaining_left = best_split["left"]
            remaining_right = best_split["right"]
        else:
            remaining_left = best_split["left"]
            remaining_right = best_split["right"]
        # Recursion
        left_tree = self._build_tree(remaining_left, depth + 1)
        right_tree = self._build_tree(remaining_right, depth + 1)
        return Node(
            left=left_tree,
            right=right_tree,
            feature=best_split["feature"],
            threshold=best_split["threshold"],
            parent_gini=parent_gini,
            num_samples=num_samples
        )

    def _predict_row(self, node, row):
        '''
        recursively predict the row
        params node: the node
        params row: the row
        return: the prediction
```

```
        '''
        if node.is_leaf():
            return node.label
        else:
            if row[node.feature] <= node.threshold:
                return self._predict_row(node.left, row)
            else:
                return self._predict_row(node.right, row)

    def _count_leaves(self, node):
        '''Helper function to count the number of leaves in a subtree'''
        if node.is_leaf():
            return 1
        else:
            return self._count_leaves(node.left) + self._count_leaves(node.right)

    def _prune_tree(self, data, node, ccp_alpha):
        """
        Recursively prune the tree based on cost complexity pruning.

        Cost(T) = R(T) + α * |T|
        R(T): sum of (parent_gini * num_samples) over all leaves of subtree T
        |T|: number of leaves in subtree T

        For a node:
        - Cost_subtree = R(subtree) + α * (leaves_subtree)
        - Cost_leaf = (node.parent_gini * node.num_samples) + α * 1

        If Cost_leaf <= Cost_subtree, we prune (make node a leaf).
        """

        # If leaf, nothing to prune
        if node.is_leaf():
            return

        # Compute the cost of the current subtree
        leaves_subtree = self._count_leaves(node)
        R_t = node.parent_gini * node.num_samples / len(data)

        # Compute the cost if we prune this node into a leaf
        R_Tt = self._subtree_impurity(node)

        # Compare to decide if we prune
        # print(node.parent_gini, node.num_samples, len(data), R_Tt)
        # print(ccp_alpha > (R_t - R_Tt) / (leaves_subtree - 1))
```

```
        if leaves_subtree == 1:
            return
        if ccp_alpha > (R_t - R_Tt) / (leaves_subtree - 1):
            label_counts = self._majority_class_in_subtree(node)
            majority_label = max(label_counts, key=label_counts.get)
            node.label = majority_label
            node.left = None
            node.right = None
            node.feature = None
            node.threshold = None

        if node.left is not None and node.right is not None:
            self._prune_tree(data, node.left, ccp_alpha)
            self._prune_tree(data, node.right, ccp_alpha)

    def _subtree_impurity(self, node):
        '''
        Helper function to compute the impurity of a subtree.
        '''
        if node.is_leaf():
            return node.parent_gini * node.num_samples
        else:
            return self._subtree_impurity(node.left) + self._subtree_impurity(node.right)

    def _majority_class_in_subtree(self, node):
        """
        Helper function to find the majority class in the subtree rooted at the given node
        Returns the label_count dict to the prune method.
        """
        if node.is_leaf():
            # Return a dictionary with the label and its count
            return {node.label: node.num_samples}

        else:
            # Initialize an empty dictionary to store counts
            label_counts = {}

            # Collect label counts from the left and right children recursively
            if node.left:
                left_counts = self._majority_class_in_subtree(node.left)
                for label, count in left_counts.items():
                    label_counts[label] = label_counts.get(label, 0) + count

            if node.right:
                right_counts = self._majority_class_in_subtree(node.right)
```

```
                for label, count in right_counts.items():
                    label_counts[label] = label_counts.get(label, 0) + count
            return label_counts
```

Check Model Section:

Unit Tests:

In [88]:
```
# Unit Tests here:
# Tests for Node
def test_Node_1():
    # Check if the node is a leaf node
    leaf_node = Node(label=1)
    assert leaf_node.is_leaf(), "Leaf node should be a leaf."
    # Check if the node is a decision node
    decision_node = Node(left="LeftNode", right="RightNode", feature=2, threshold=0.5)
    assert not decision_node.is_leaf(), "Decision node should not be a leaf."
    print("Node tests 1 passed.")

def test_Node_2():
    # Check if the node is a leaf node
    leaf_node = Node(label="hellow")
    assert leaf_node.is_leaf(), "Leaf node should be a leaf."
    # Check if the node is a decision node
    decision_node = Node(left="LeftNode", right="RightNode", feature=2, threshold=0.5)
    assert not decision_node.is_leaf(), "Decision node should not be a leaf."
    print("Node tests 2 passed.")

def test_fit_1():
    data = np.array([[1, 0], [2, 1], [3, 0], [4, 1]])
    cart = CART(max_depth=1)
    cart.fit(data)
    print("test fit 1 passed")
    assert cart.tree is not None, "Tree should not be None after fitting."

def test_fit_2():
    data = np.array([]).reshape(0, 8)
    cart = CART()
    try:
        cart.fit(data)
        assert False, "Fitting empty data should raise an error."
    except:
```

```
        pass
    print("test fit 2 passed")

def test_loss_1():
    data = np.array([[1, 0], [2, 1], [3, 0], [4, 1]])
    cart = CART(max_depth=4)
    cart.fit(data)
    loss = cart.loss(data)
    assert loss == 0, "Loss calculation is incorrect."
    print("test loss 1 passed")

def test_loss_2():
    train_data = np.array([[1, 0], [2, 1]])
    test_data = np.array([[3, 0], [4, 1]])
    cart = CART(max_depth=1)
    cart.fit(train_data)
    loss = cart.loss(test_data)
    assert loss == 0.5, "Loss calculation is incorrect."
    print("test loss 2 passed")

def test_accuracy_1():
    data = np.array([[1, 0], [2, 1], [3, 0], [4, 1]])
    cart = CART(max_depth=4)
    cart.fit(data)
    acc = cart.accuracy(data)
    assert acc == 1, "Accuracy calculation is incorrect."
    print("test accuracy 1 passed")

def test_accuracy_2():
    train_data = np.array([[1, 0], [2, 1]])
    test_data = np.array([[3, 0], [4, 1]])
    cart = CART(max_depth=1)
    cart.fit(train_data)
    acc = cart.accuracy(test_data)
    assert acc == 0.5, "Accuracy calculation is incorrect."
    print("test accuracy 2 passed")

# Tests for loss and accuracy
def test_loss_acc():
    tree = Node(left=Node(label=1), right=Node(label=0), feature=0, threshold=1.5)
    cart = CART()
    cart.tree = tree
    data = np.array([
        [1, 2, 1],
        [2, 3, 0],
```

```
        [0.5, 1, 1],
        [3, 4, 0]
    ])
    # Check if loss = 0 and accuracy = 1
    assert cart.loss(data) == 0, "Loss calculation is incorrect."
    assert cart.accuracy(data) == 1, "Accuracy calculation is incorrect."
    print("Loss and accuracy tests passed.")

def test_predict_1():
    # Check if a single row prediction is correct
    tree = Node(left=Node(label=1), right=Node(label=0), feature=0, threshold=1.5)
    cart = CART()
    row = np.array([1, 2])  # Expected: left -> 1
    pred = cart._predict_row(tree, row)
    assert pred == 1, "Prediction for single row is incorrect."
    # Check predictions for a dataset
    cart.tree = tree
    data = np.array([
        [1, 2],
        [2, 3]
    ])
    preds = cart.predict(data)
    assert np.array_equal(preds, [1, 0]), "Batch predictions are incorrect."
    print("Predict test 1 passed.")

def test_predict_2():
    train_data = np.array([[1, 0],
                           [2, 1]])
    test_data = np.array([[6], [101]])
    cart = CART(max_depth=2)
    cart.fit(train_data)
    try:
        cart.predict(test_data)
        assert False, "Predictions should raise an error if test data has different number of features."
    except:
        pass
    print("Predict test 2 passed.")

# Tests for _find_best_split
def test_find_best_split():
    data = np.array([
        [1, 2.5, 0],
        [2, 3.5, 1],
        [1.5, 2, 0]
    ])
```

```
    cart = CART()
    best_split = cart._find_best_split(data)
    # Check a valid split is found and the split is correct
    # Should not split on the target but split on one of the continuous features
    assert best_split is not None, "Best split should not be None."
    assert best_split["type"] == "continuous", "Best split should be continuous."
    assert best_split["feature"] != 2, "Best split should not be on the target."
    data = np.array([
        [1, 2.5, 0],
        [1, 2.5, 0],
        [1, 2.5, 0]
    ])
    cart = CART()
    best_split = cart._find_best_split(data)
    # Check that no split should be found if all features are the same
    assert best_split is None, "Best split should be None."
    print("Find best split tests passed.")

# Tests for Gini impurity calculations
def test_gini_1():
    data = np.array([
        [1, 2.5, 0],
        [2, 3.5, 1]
    ])
    cart = CART()
    gini = cart._gini_for_node(data)
    assert abs(gini - 0.5) < 1e-6, "Gini impurity for node is incorrect."
    left = data[:1]
    right = data[1:]
    gini = cart._gini_for_split(data, left, right)
    assert abs(gini - 0) < 1e-6, "Gini impurity for split is incorrect."
    print("Gini test 1 passed.")

def test_gini_2():
    data = np.array([
        [1, 0], [2, 0]
    ])
    cart = CART()
    gini = cart._gini_for_node(data)
    assert gini == 0, "Gini impurity for node is incorrect."
    left = data[:1]
    right = data[1:]
    gini = cart._gini_for_split(data, left, right)
    assert abs(gini) < 1e-9, "Gini impurity for split is incorrect."
    print("Gini test 2 passed.")
```

```
def test_split_1():
    data = np.array([
        [1, 2.5, 0],
        [1.5, 2, 0],
        [2, 3.5, 1]
    ])
    cart = CART()
    left, right = cart._split(data, 0, 1.25)
    # Check if the split is correct
    assert len(left) == 1, "Left split should have 1 row."
    assert len(right) == 2, "Right split should have 2 rows."
    print("Split test 1 passed.")

def test_split_2():
    data = np.array([
        [1, 2.5, 0],
        [1.5, 2, 0],
        [2, 3.5, 1]
    ])
    cart = CART()
    left, right = cart._split(data, 1, -1)
    # Check if the split is correct
    assert len(left) == 0, "Left split should have 0 rows."
    assert len(right) == 3, "Right split should have 3 row."
    print("Split test 2 passed.")

def test_best_split_1():
    data = np.array([
        [1, 2.5, 0],
        [1.5, 2, 0],
        [2, 3.5, 1]
    ])
    cart = CART()
    best_gain, best_split = cart._find_best_split(data)
    best_gain = round(best_gain, 5)
    assert best_gain == 0.44444, "Best gain is incorrect."
    assert best_split["feature"] == 1, "Best split feature is incorrect."
    assert best_split["threshold"] == 3, "Best split threshold is incorrect."
    print("Best split test 1 passed.")

def test_best_split_2():
    data = np.array([
        [2, 2, 0],
        [2.5, 2.5, 0],
```

```
        [3.5, 3.5, 0]
    ])
    cart = CART()
    best_gain, best_split = cart._find_best_split(data)
    best_gain = round(best_gain, 5)
    assert best_gain == 0, "Best gain is incorrect."
    assert best_split["feature"] == 1, "Best split feature is incorrect."
    assert best_split["threshold"] == 3, "Best split threshold is incorrect."
    print("Best split test 2 passed.")

def test_majority_class_1():
    data = np.array([
        [1, 2.5, 0],
        [1.5, 2, 0],
        [2, 3.5, 1]
    ])
    cart = CART()
    majority = cart._majority_class(data)
    assert majority == 0, "Majority class is incorrect."
    print("Majority class test 1 passed.")

def test_majority_class_2():
    data = np.array([
        [1, 2.5, 1],
        [1.5, 2, 1],
        [2, 3.5, 1]
    ])
    cart = CART()
    majority = cart._majority_class(data)
    assert majority == 1, "Majority class is incorrect."
    print("Majority class test 2 passed.")

def test_build_tree_1():
    data = np.array([
        [1, 2.5, 0],
        [1.5, 2, 0],
        [2, 3.5, 1],
    ])
    cart = CART(max_depth=2)
    tree = cart._build_tree(data)
    # Check if the tree is built correctly
    assert tree.feature == 1, "Root feature is incorrect."
    assert tree.threshold == 3, "Root threshold is incorrect."
    assert tree.left.label == 0, "Left leaf label is incorrect."
    assert tree.right.label == 1, "Right leaf label is incorrect."
```

```
    print("Build tree test 1 passed.")

def test_build_tree_2():
    data = np.array([
        [1, 2.5, 0],
        [2, 3.5, 0],
        [1.5, 2, 0]
    ])
    cart = CART(max_depth=2)
    tree = cart._build_tree(data)
    # Check if the tree is built correctly
    assert tree.label == 0, "Root label is incorrect."
    print("Build tree test 2 passed.")
```

In [89]:
```
# Below two functions are testing the Node to take numerical and string labels, and they should be leaf node
# Edge case: numerical and string labels
test_Node_1()
test_Node_2()

# Below two functionare testing the fit function. Tets if fit builds a tree from small dataset or empty data
# Edge case: small and empty dataset
test_fit_1()
test_fit_2()

# Below functions are testing the loss function.
# Edge case: loss function with 0 loss and 0.5 loss and multi-dimensional data
test_loss_1()
test_loss_2()
test_loss_acc() # this test both loss and accuracy for multi-dimensional data

# Below functions are testing the accuracy function.
# Edge case: accuracy function with 1 accuracy and 0.5 accuracy
test_accuracy_1()
test_accuracy_2()

# testing for predict
# Edge case: single row prediction and batch prediction and unseen data
test_predict_1()
test_predict_2()

# Below functionare testing the _gini_for_node and _gini_for_split function.
# Edge case: gini for node and split with 0 gini and multi-dimensional data
test_gini_1()
test_gini_2()
```

```
# Below function is testing the _split function.
# Edge case: split function with 1 row and 0 row, with a threshold being negative or below every value
test_split_1()
test_split_2()

# Below function is testing the _find_best_split function.
# Edge case: best split with 0 gain and 0.44444 gain, and there should be a tie, choose later feature.
test_best_split_1()
test_best_split_2()

# Below function are for majority class
# Edge case: multiple class vs a single class
test_majority_class_1()
test_majority_class_2()

# Below functions are for testing build_tree
# Edge case: build a simple tree with depth 2 and a tree with all the same features
# also tested if the data is not in order
test_build_tree_1()
test_build_tree_2()
```

```
Node tests 1 passed.
Node tests 2 passed.
test fit 1 passed
test fit 2 passed
test loss 1 passed
test loss 2 passed
Loss and accuracy tests passed.
test accuracy 1 passed
test accuracy 2 passed
Predict test 1 passed.
Predict test 2 passed.
Gini test 1 passed.
Gini test 2 passed.
Split test 1 passed.
Split test 2 passed.
Best split test 1 passed.
Best split test 2 passed.
Majority class test 1 passed.
Majority class test 2 passed.
Build tree test 1 passed.
Build tree test 2 passed.
```

We don't need to unit test for prune because those two are a function that calss _prune. So we can just test those functions instead. And predict_row and count leaves are not tested because they are just function that recursively gets the label or could

leaves number.

In [7]:
```python
def visualize_tree(tree, size=(10, 10)):
    """
    Visualize the decision tree
    param tree: the decision tree
    return: the graph of the decision tree
    additional reference: https://graphviz.readthedocs.io/en/stable/manual.html
    """
    dot = Digraph(format="png")
    dot.attr(size=str(size))
    def add_nodes_edges(node, dot, counter):
        """
        Add nodes and edges to the graph
        param node: the node of the decision tree
        param dot: the graph
        param counter: the counter
        return: the counter, which serves as the node id, required
        """
        if node is None:
            return counter # Check if the node is None
        node_id = str(counter)

        if node.is_leaf():
            label = f"Label: {node.label}\nGini: {node.parent_gini:.3f}\nSamples: {node.num_samples}"
        else:
            label = (f"Feature {node.feature} ≤ {node.threshold:.2f}\n"
                     f"Gini: {node.parent_gini:.3f}\n"
                     f"Samples: {node.num_samples}")
        # display the node
        dot.node(node_id, label) # add node to the graph
        counter += 1 # increment the counter which is the node id

        if node.left:
            left_id = str(counter)
            counter = add_nodes_edges(node.left, dot, counter)
            dot.edge(node_id, left_id, label="True")
        if node.right:
            right_id = str(counter)
            counter = add_nodes_edges(node.right, dot, counter)
            dot.edge(node_id, right_id, label="False")
        # recursively add nodes and edges
        return counter
```

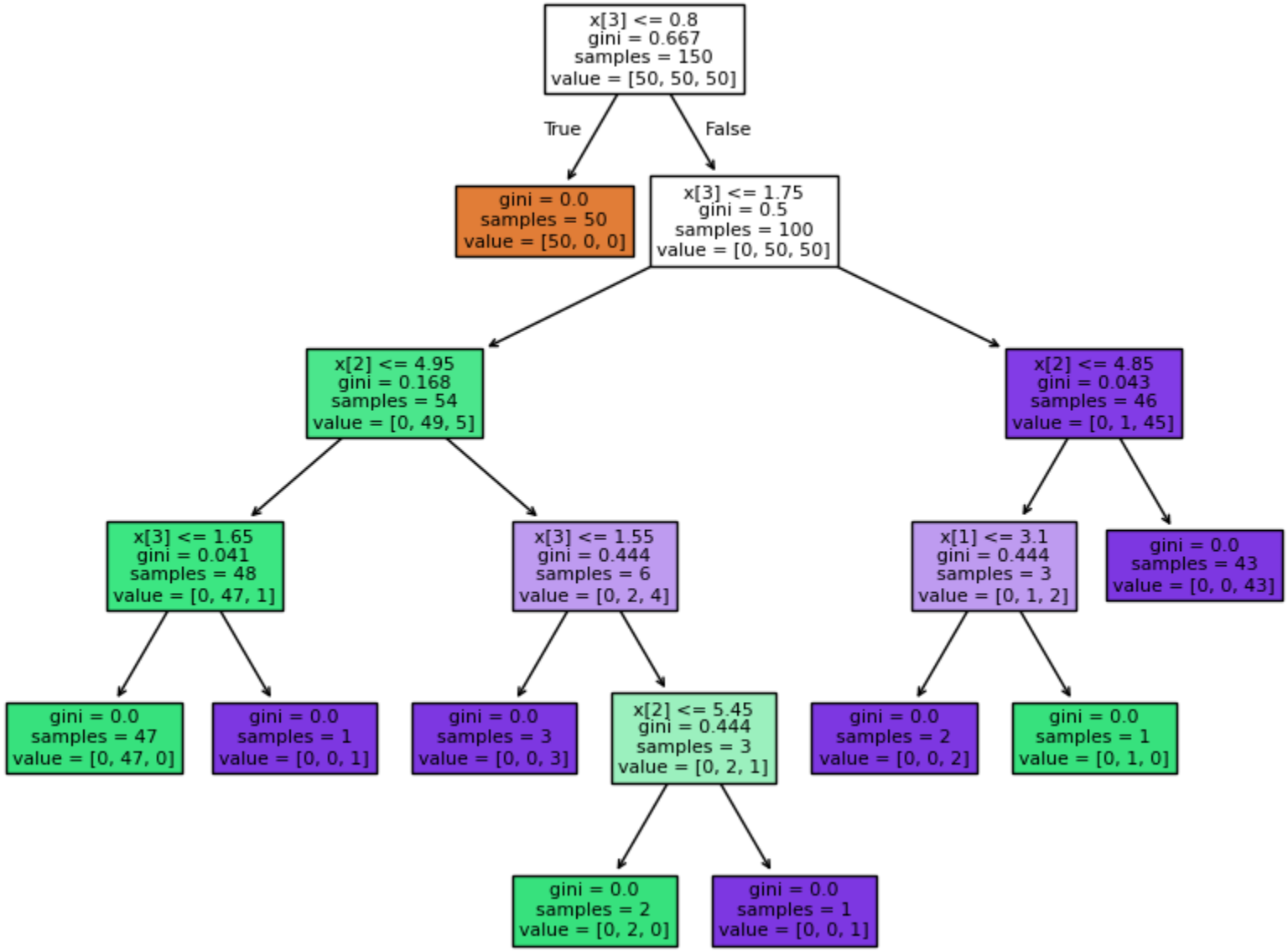
```
    add_nodes_edges(tree.tree, dot, 0)
    return dot # get the graph and return
```

Testing iris, a previous work implementation

input the entire dataset as training!

In [8]:
```
iris = load_iris() # use the iris dataset from sklearn
X = iris.data
y = iris.target
iris_data = np.column_stack((X, y))
```

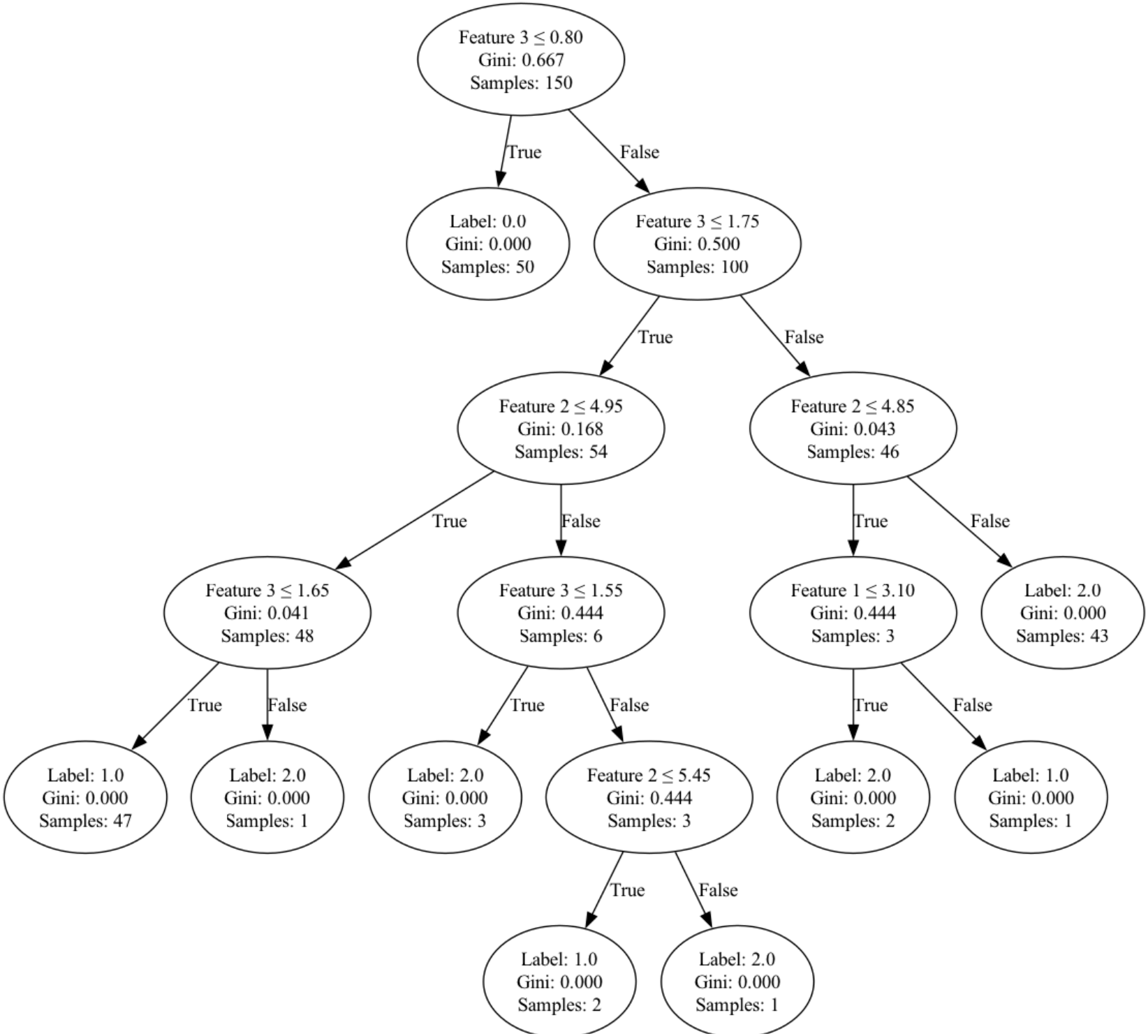
In [9]:
```
# sklearn's decision tree structure for reference
clf = tree.DecisionTreeClassifier(random_state=0)
clf = clf.fit(X, y)
fig = plt.figure(figsize=(10, 8))
tree.plot_tree(clf, filled=True)
plt.savefig("../graph/sklearn_iris.png") # save to the folder for reference
plt.show()
```



In [10]:
```
# our implementation
# we set max_depth=5 to match the sklearn tree and min_samples_split=2 to match the default value in sklearn
cart = CART(max_depth=5, min_samples_split=2)
cart.fit(iris_data)
```

```
tree_graph = visualize_tree(cart)
tree_graph.render('../graph/CART_iris', format='png', cleanup=True) # save to the graph folder to better org
```

Out[10]:
```
'../graph/CART_iris.png'
```


Feature 3 ≤ 0.80
Gini: 0.667
Samples: 150
True
False
Label: 0.0
Gini: 0.000
Samples: 50
Feature 3 ≤ 1.75
Gini: 0.500
Samples: 100
True
False
Feature 2 ≤ 4.95
Gini: 0.168
Samples: 54
Feature 2 ≤ 4.85
Gini: 0.043
Samples: 46
True
False
True
False
Feature 3 ≤ 1.65
Gini: 0.041
Samples: 48
Feature 3 ≤ 1.55
Gini: 0.444
Samples: 6
Feature 1 ≤ 3.10
Gini: 0.444
Samples: 3
Label: 2.0
Gini: 0.000
Samples: 43
True
False
True
False
True
False
Label: 1.0
Gini: 0.000
Samples: 47
Label: 2.0
Gini: 0.000
Samples: 1
Label: 2.0
Gini: 0.000
Samples: 3
Feature 2 ≤ 5.45
Gini: 0.444
Samples: 3
Label: 2.0
Gini: 0.000
Samples: 2
Label: 1.0
Gini: 0.000
Samples: 1
True
False
Label: 1.0
Gini: 0.000
Samples: 2
Label: 2.0
Gini: 0.000
Samples: 1

In [11]:
```
iris = load_iris()
X = iris.data
y = iris.target
data = np.column_stack((X, y))
train_data, test_data = train_test_split(data, test_size=0.2, random_state=0)
model = CART(max_depth=3, min_samples_split=10, random_state=0)
model.fit(train_data)
train_accuracy = model.accuracy(train_data)
test_accuracy = model.accuracy(test_data)
print("For the Iris dataset, here's the accuracy with CART")
print(f"Training accuracy: {train_accuracy:.3f}")
print(f"Testing accuracy: {test_accuracy:.3f}")
```

```
For the Iris dataset, here's the accuracy with CART
Training accuracy: 0.967
Testing accuracy: 0.967
```

In [12]:
```
iris = load_iris()
X = iris.data
y = iris.target
data = np.column_stack((X, y))
train_data, test_data = train_test_split(data, test_size=0.2, random_state=0)
X_train = train_data[:, :-1]
y_train = train_data[:, -1]
X_test = test_data[:, :-1]
y_test = test_data[:, -1]
model = DecisionTreeClassifier(max_depth=3, min_samples_split=10, random_state=0)
model.fit(X_train, y_train)
train_accuracy = accuracy_score(y_train, model.predict(X_train))
test_accuracy = accuracy_score(y_test, model.predict(X_test))
print("For the Iris dataset, here's the accuracy with sklearn's DecisionTreeClassifier")
print(f"Training accuracy: {train_accuracy:.3f}")
print(f"Testing accuracy: {test_accuracy:.3f}")
```

```
For the Iris dataset, here's the accuracy with sklearn's DecisionTreeClassifier
Training accuracy: 0.967
Testing accuracy: 0.967
```

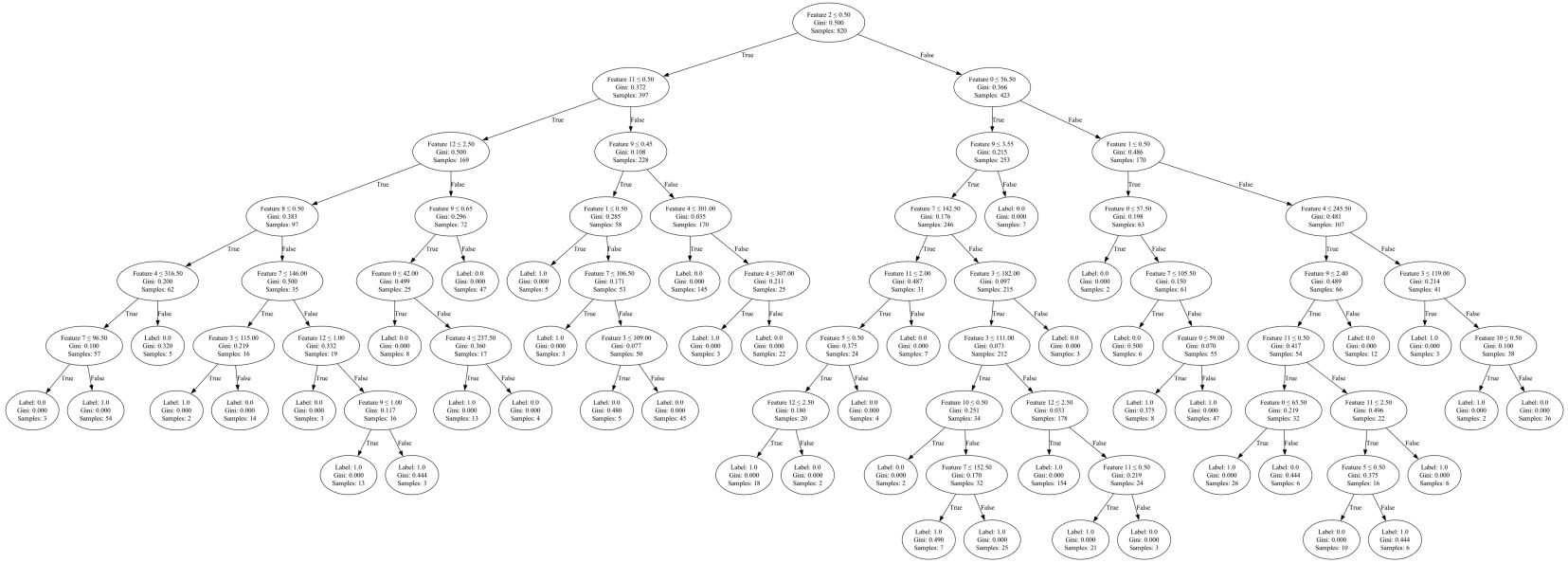
Check iris accuracy between CART and scikitlearn

Using heart.csv

In [13]:
```
data = np.loadtxt("../data/heart.csv", delimiter=",", skiprows=1)
#train_data, val_data, test_data = train_test_split(data, test_size=0.4, random_state=0)
train_data, test_data = train_test_split(data, test_size=0.2, random_state=0)
#np.savetxt("../data/training_heart.csv", train_data, delimiter=",")
model = CART(max_depth=8, min_samples_split=10, random_state=0)
model.fit(train_data)
train_accuracy = model.accuracy(train_data)
test_accuracy = model.accuracy(test_data)
print("For the Heart dataset, here's the accuracy with CART")
print(f"Training accuracy: {train_accuracy:.3f}")
print(f"Testing accuracy: {test_accuracy:.3f}")
tree_graph = visualize_tree(model, size=(12,8))
tree_graph.render('../graph/CART_heart', format='png', cleanup=True)
```

```
For the Heart dataset, here's the accuracy with CART
Training accuracy: 0.980
Testing accuracy: 0.946
```

Out[13]:
```
'../graph/CART_heart.png'
```



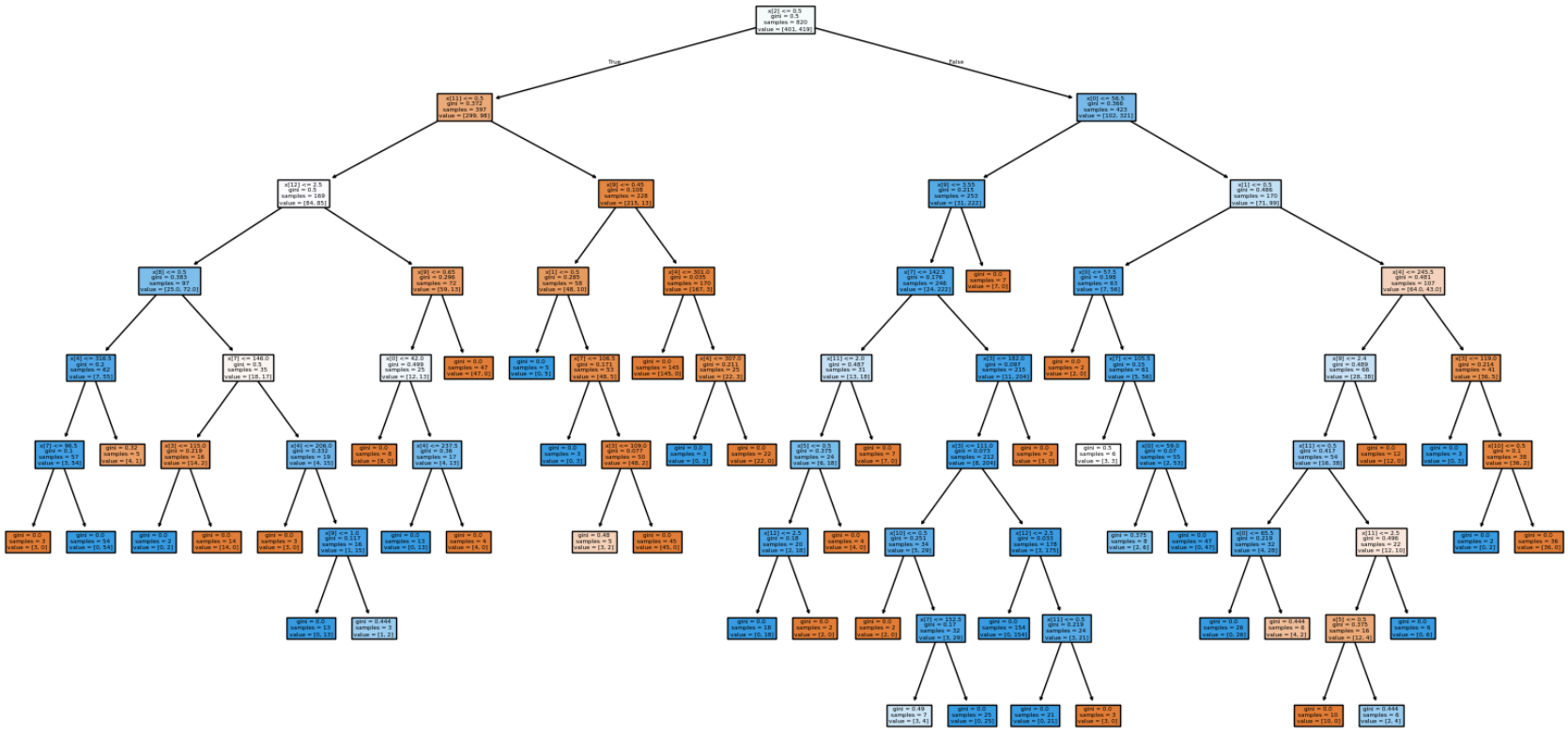
In [14]:
```
model = DecisionTreeClassifier(max_depth=8, min_samples_split=10, random_state=0)
train_data, test_data = train_test_split(data, test_size=0.2, random_state=0)
X_train = train_data[:, :-1]
y_train = train_data[:, -1]
X_test = test_data[:, :-1]
```

```
y_test = test_data[:, -1]
model.fit(X_train, y_train)

# Predict on the training and testing datasets
train_preds = model.predict(X_train)
test_preds = model.predict(X_test)

# Compute the accuracies
train_accuracy = accuracy_score(y_train, train_preds)
test_accuracy = accuracy_score(y_test, test_preds)

# Print the results
print("For the Heart dataset, here's the accuracy with sklearn's DecisionTreeClassifier")
print(f"Training accuracy: {train_accuracy:.3f}")
print(f"Testing accuracy: {test_accuracy:.3f}")
```

```
For the Heart dataset, here's the accuracy with sklearn's DecisionTreeClassifier
Training accuracy: 0.980
Testing accuracy: 0.946
```

In [15]:

```
plt.figure(figsize=(20, 10))
plot_tree(model, filled=True)
plt.savefig("../graph/sklearn_heart.png")
```



As you can see the accuracy and tree structure looks exactly the same. We did a lot of work to reproduce the tree, such as setting Python and numpy's random_state to the same seed, also used sklearn's train-test-split to guarantee a same split. You can also play around with the depth, the accuracy should be the same as well.

Pruned results

Pruned Scikitlearn implementation:

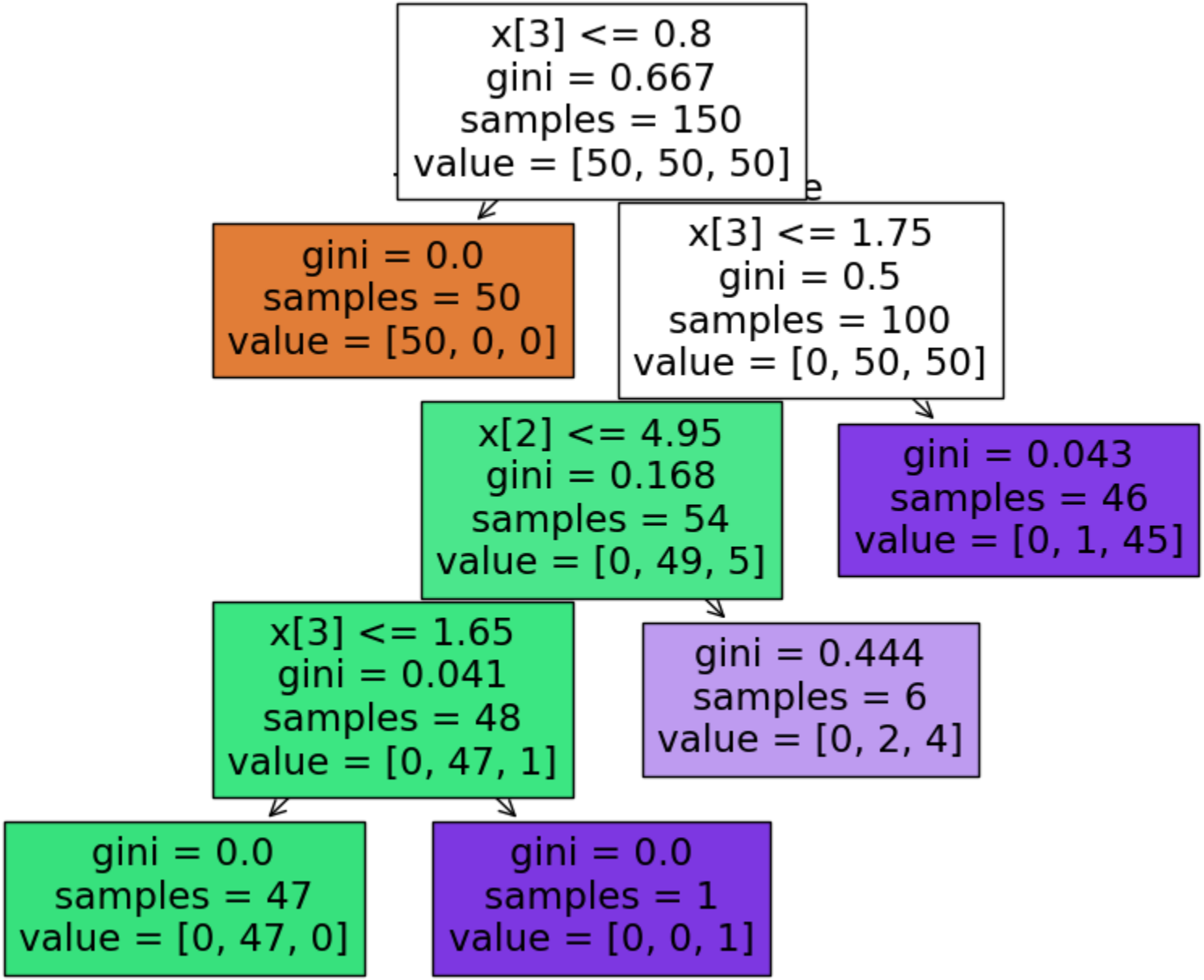
In [21]:
```
def manual_cross_val_score(model, X, y, cv=5, random_state = 0, model_type="SKL"):
    '''
    Perform cross validation for the model
    params model: the model
    params X: the features
    params y: the labels
```

```
    params cv: the number of folds
    params random_state: the random state
    params model_type: the model type (skl or cart)
    '''
    kf = KFold(n_splits=cv, random_state=random_state, shuffle=True)
    scores = []
    for train_i, val_i in kf.split(X):
        train_X, val_X = X[train_i], X[val_i]
        train_y, val_y = y[train_i], y[val_i]
        if model_type == "SKL":
            clone_model = DecisionTreeClassifier(random_state=model.random_state, ccp_alpha=model.ccp_alpha)
            clone_model.fit(train_X, train_y)
            val_preds = clone_model.predict(val_X)
            score = accuracy_score(val_y, val_preds)
        else:
            clone_model = CART(random_state=model.random_state, ccp_alpha=model.ccp_alpha)
            clone_model.fit(np.column_stack((train_X, train_y)))
            val_preds = clone_model.predict(np.column_stack((val_X, val_y)))
            score = accuracy_score(val_y, val_preds)
        scores.append(score)
    return scores
```

For Iris

In [90]:
```
iris = load_iris() # use the iris dataset from sklearn
X = iris.data
y = iris.target
iris_data = np.column_stack((X, y))
# sklearn's decision tree structure for reference
clf = tree.DecisionTreeClassifier(random_state=0, ccp_alpha=0.01)
clf = clf.fit(X, y)
print("Accuracy of the sklearn decision tree:", accuracy_score(y, clf.predict(X)))
fig = plt.figure(figsize=(10, 8))
tree.plot_tree(clf, filled=True)
plt.savefig("../graph/sklearn_irisPrune.png") # save to the folder for reference
plt.show()
```

```
Accuracy of the sklearn decision tree: 0.98
```

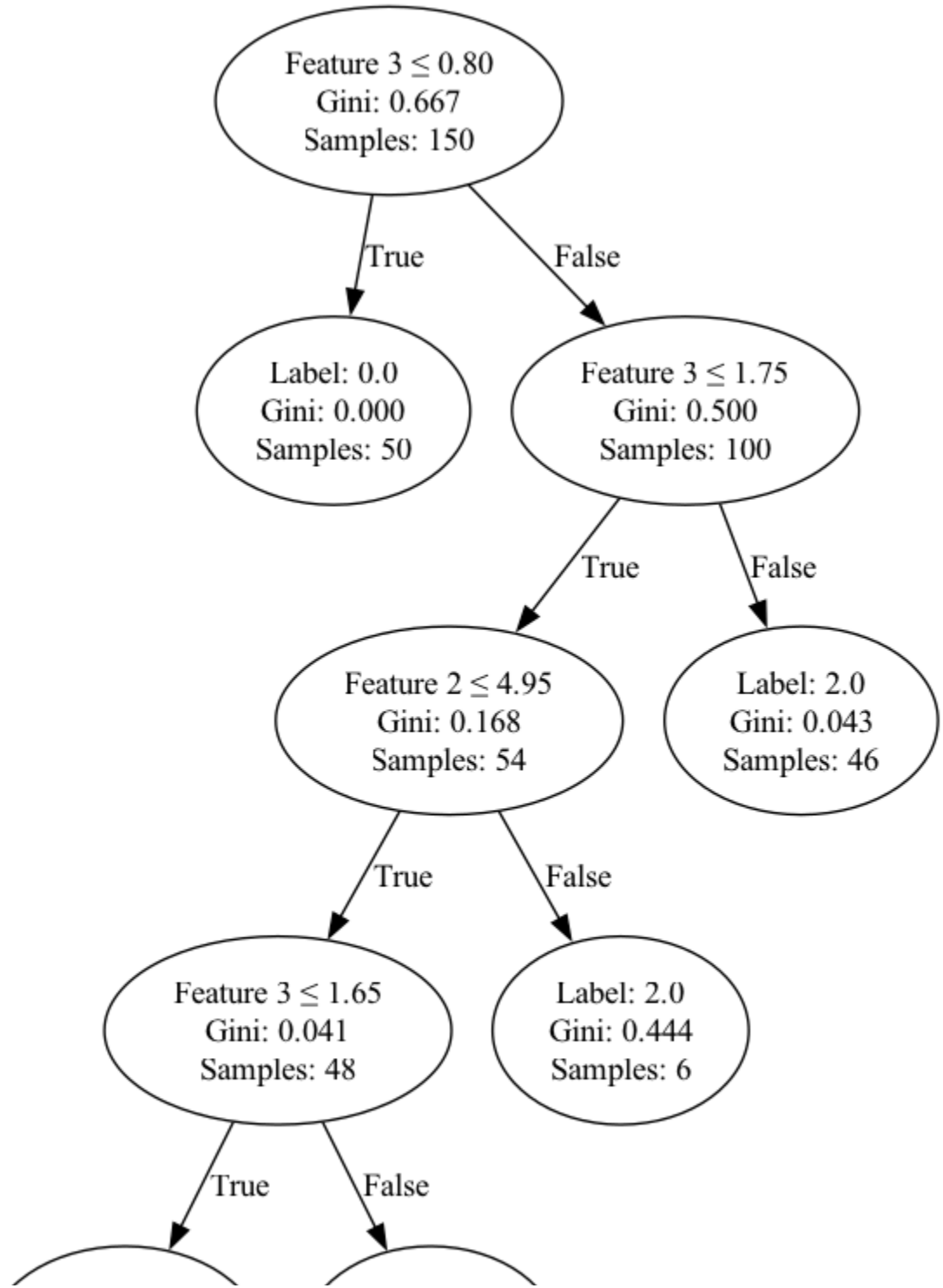


In [91]:
```
# our implementation
# we set max_depth=5 to match the sklearn tree and min_samples_split=2 to match the default value in sklearn
cart = CART(max_depth=5, min_samples_split=2, ccp_alpha=0.01, random_state=0)
cart.fit(iris_data)
cart.prune(iris_data, ccp_alpha=0.01)
```

```
print("Accuracy of the manual decision tree:", cart.accuracy(iris_data))
tree_graph = visualize_tree(cart)
tree_graph.render('../graph/CART_iris_prune', format='png', cleanup=True) # save to the graph folder to bett
```

```
Accuracy of the manual decision tree: 0.98
```

Out[91]:
```
'../graph/CART_iris_prune.png'
```


Feature 3 ≤ 0.80
Gini: 0.667
Samples: 150
True
False
Label: 0.0
Gini: 0.000
Samples: 50
Feature 3 ≤ 1.75
Gini: 0.500
Samples: 100
True
False
Feature 2 ≤ 4.95
Gini: 0.168
Samples: 54
Label: 2.0
Gini: 0.043
Samples: 46
True
False
Feature 3 ≤ 1.65
Gini: 0.041
Samples: 48
Label: 2.0
Gini: 0.444
Samples: 6
True
False

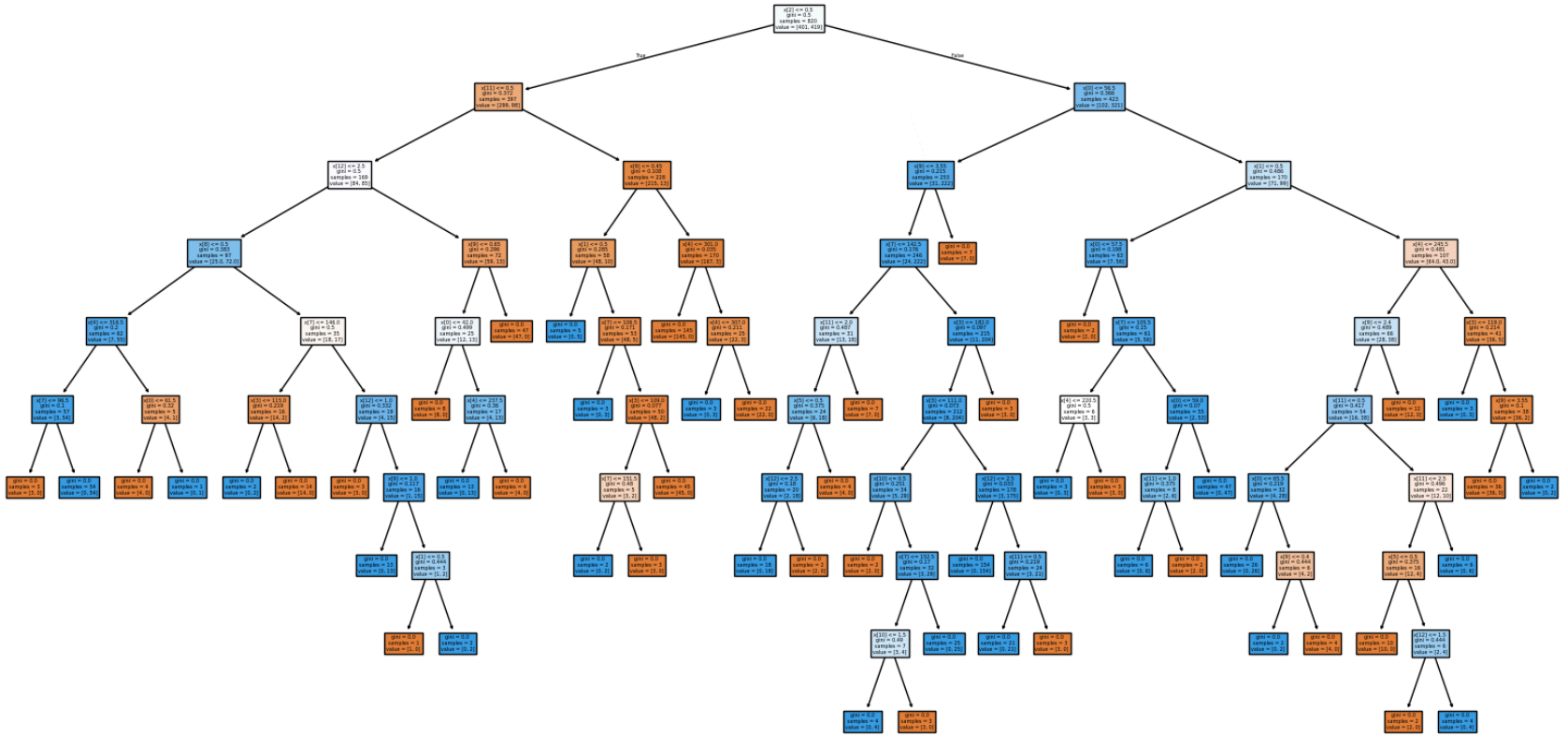
As we can see from the two graphs above, the accuracy is the same, and the tree is the same

In [99]:
```
data = np.loadtxt("../data/heart.csv", delimiter=",", skiprows=1)
train_data, test_data = train_test_split(data, test_size=0.2, random_state=0)
X_train = train_data[:, :-1]
y_train = train_data[:, -1]
X_test = test_data[:, :-1]
y_test = test_data[:, -1]
ccp_alphas = np.linspace(0.0, 0.001, 20)  # 10 values from 0.0 to 0.02

best_alpha = None
best_score = -np.inf
# try to find the best ccp_alpha
for alpha in ccp_alphas:
    model = DecisionTreeClassifier(random_state=0, ccp_alpha=alpha)
    scores = manual_cross_val_score(model, X_train, y_train, cv=5, random_state=0, model_type="SKL")
    mean_score = np.mean(scores)
    if mean_score >= best_score:
        best_score = mean_score
        best_alpha = alpha

print(f"\nBest ccp_alpha: {best_alpha:.8f} with mean CV accuracy: {best_score:.2f}")

best_model = DecisionTreeClassifier(random_state=0, ccp_alpha=0.001)
scores = manual_cross_val_score(model, X_train, y_train, cv=5)
mean_score = np.mean(scores)
print(f"Mean cross-validation score: {mean_score:.2f}")
best_model.fit(X_train, y_train)
test_preds = best_model.predict(X_test)
test_accuracy = accuracy_score(y_test, test_preds) # get the test accuracy
print(f"Test accuracy with best ccp_alpha={best_alpha:.4f}: {test_accuracy:.4f}")
```

```
Best ccp_alpha: 0.00100000 with mean CV accuracy: 0.99
Mean cross-validation score: 0.99
Test accuracy with best ccp_alpha=0.0010: 1.0000
```

In [97]:
```
plt.figure(figsize=(20, 10))
plot_tree(best_model, filled=True)
```

```
plt.savefig("../graph/sklearn_heartPrune.png")
```



CART Pruned implementation

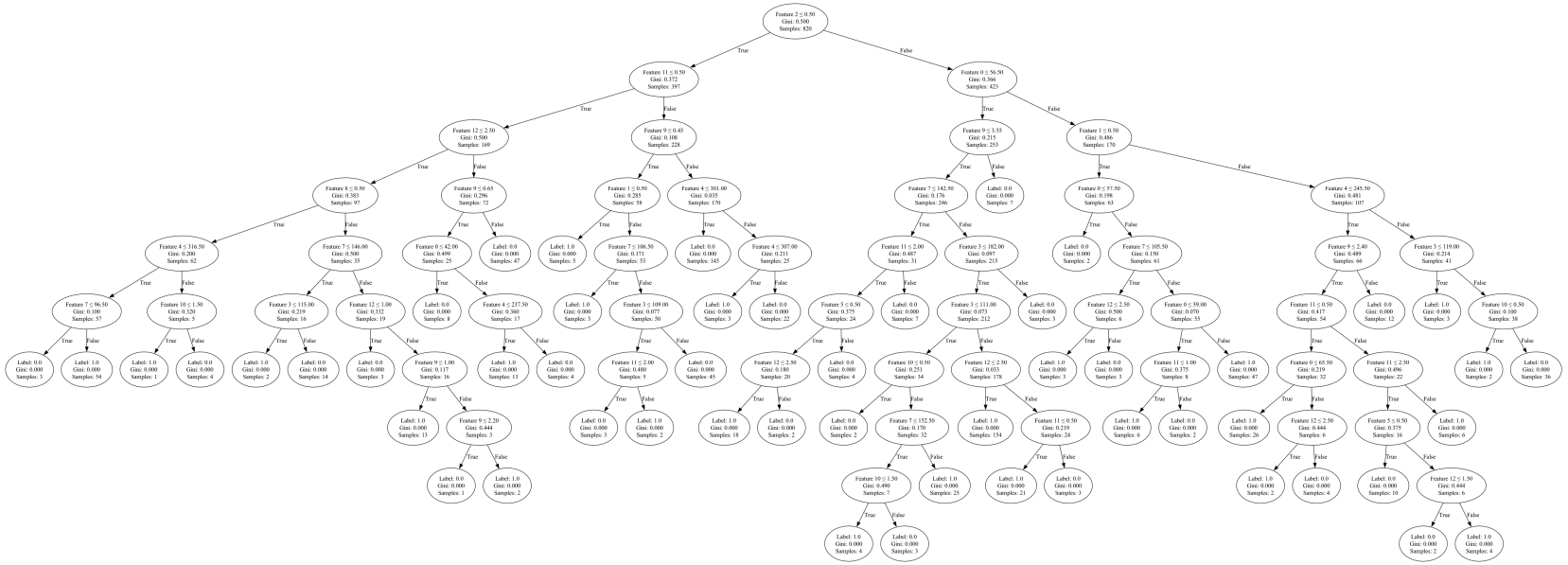
In [100…

```
data = np.loadtxt("../data/heart.csv", delimiter=",", skiprows=1)
train_data, test_data = train_test_split(data, test_size=0.2, random_state=0)
X_train = train_data[:, :-1]
y_train = train_data[:, -1]
X_test = test_data[:, :-1]
y_test = test_data[:, -1]

model = CART(random_state=0, ccp_alpha=0.001)
scores = manual_cross_val_score(model, X_train, y_train, cv=5, random_state=0, model_type="CART")
mean_score = np.mean(scores)
print(f"Mean cross-validation score: {mean_score:.2f}")
model.fit(np.column_stack((X_train, y_train)))
test_preds = best_model.predict(X_test)
test_accuracy = accuracy_score(y_test, test_preds)
```

```
print(f"Test accuracy with best ccp_alpha={best_alpha:.4f}: {test_accuracy:.4f}")

tree_graph = visualize_tree(model, size=(12,8))
tree_graph.render('../graph/CART_heartPrune', format='png', cleanup=True)
```

```
Mean cross-validation score: 0.99
Test accuracy with best ccp_alpha=0.0010: 1.0000
```

Out[100]: '../graph/CART_heartPrune.png'



We applied 5-fold to the heart data because it is more complicated than Iris. Despite coming up with our own implementation of pruning, we still achieved scikit-learn's result, the testing accuracy is going up compared to the implementation without pruning. The reason why this tree appears to be deeper than the ones above is because we didn't set a max_depth for the pruned result for both scikit-learn and our implementation, so the resulting graphs are larger, and because we compared our graph to the scikit-learn generated graph, we can say that we achieved the correct result. As we can see that our accuracy between not pruning and pruning is very small, which suggest that our model did not overfit, the reason might be because the tree is not a complex tree and it does well in generalizing the dataset.

Result Overview

As we can see from the above codes, we achieved the same training and testing accuracy and the same tree structure for both dataset (Iris and heart) with both Scikit-learn and our implementation. Then we tried the implementation with pruning optimization, we achieved better result compared to not pruned.

Updated accuracy:

Without Pruning

Iris training CART: 0.967

Iris testing CART: 0.967

Iris training Scikit-learn: 0.967

Iris testing Scikit-learn: 0.967

Heart training CART: 0.98

Heart testing CART: 0.946

Heart training Scikit-learn: 0.98

Heart testing Scikit-learn: 0.946

With Pruning

Iris testing CART: 0.98

Iris testing Scikit-learn: 0.98

Heart average CART: 0.99

Heart testing CART: 1

Heart average Scikit-learn: 0.99

Heart testing Scikit-learn: 1

Fun fact

Our team name is UC Providence because we all graduated from the University of California, though from different campuses. Yixun graduated from UC Santa Barbara, David graduated from UC Irvine, and Liang graduated from UC San Diego. Interestingly, We didn't know we all went to the UC schools until our first meeting, and that's how we came up with the name.

References:

1. Fisher, R. (1936). Iris [Dataset]. UCI Machine Learning Repository. Available at: https://doi.org/10.24432/C56C76/ (Accessed: 1 December 2024).
2. Janosi, A., Steinbrunn, W., Pfisterer, M., & Detrano, R. (1989). Heart Disease [Dataset]. UCI Machine Learning Repository. Available at: https://doi.org/10.24432/C52P4X/ (Accessed: 1 December 2024).
3. Scikit-learn, 2024. Plot Iris Dataset. Available at: https://scikit-learn.org/1.5/auto_examples/datasets/plot_iris_dataset.html/ (Accessed: 1 December 2024)
4. Scikit-learn, 2024. Plot Iris Decision Tree Classifier. Available at: https://scikit-learn.org/1.5/auto_examples/tree/plot_iris_dtc.html (Accessed 1 December 2024).
5. Scikit-learn, 2024. DecisionTreeClassifier. Available at: https://scikit-learn.org/1.5/modules/generated/sklearn.tree.DecisionTreeClassifier.html (Accessed 1 December 2024).
6. meanxai, 2023. [MXML-2-09] Decision Trees [9/11] - CART, Cost Complexity Pruning (CCP). YouTube. Available at: https://www.youtube.com/watch?v=my3IjAS5UUM&t=845s (Accessed: 12 December 2024).

DATA2060 - final project contribution form

Your response has been recorded.

Submit another response