project

December 6, 2025

0.1 Part 1. Overview of CART (Classification And Regression Trees) for Classification

0.2 Introduction

CART (Classification And Regression Trees) is a decision tree algorithm developed by Breiman et al. (1984) that builds binary trees for both classification and regression tasks. Unlike ID3, which we covered in class, CART uses different splitting criteria and creates strictly binary trees. In this project, we implement CART for classification problems. Modern toolkits such as scikit-learn adopt the same defaults (Pedregosa et al., 2011), so matching their behavior provides a strong sanity check (Breiman et al., 1984; Loh, 2011).

0.3 Representation

CART represents a classification model as a binary decision tree where: - Each **internal node** represents a decision based on a single feature and threshold - Each **edge** represents the outcome of the decision (left child for values threshold, right for values > threshold)
- Each **leaf node** represents a class prediction

For a given input $\mathbf{x} = (x_1, x_2, ..., x_d)$, the prediction is made by: 1. Starting at the root node 2. Following decision rules down the tree: if $x_j \leq t$ go left, else go right 3. Returning the majority class at the reached leaf node

Mathematically, the tree partitions the feature space into disjoint regions $R_1, R_2, ..., R_M$ and predicts:

$$\hat{y}(\mathbf{x}) = \sum_{m=1}^{M} c_m \cdot \mathbb{1}(\mathbf{x} \in R_m)$$

where c_m is the majority class in region R_m.

0.4 Loss Function: Gini Impurity

CART uses the **Gini impurity** as the default criterion to measure the quality of a split. For a node containing samples from classes $1, 2, ..., K$, the Gini impurity is:

$$G = 1 - \sum_{k=1}^{K} p_k^2$$

where p_k is the proportion of samples belonging to class k in the node.

Properties: - $G = 0$ when all samples belong to one class (pure node) - G is maximized when classes are equally distributed - For binary classification: $G = 2p(1-p)$ where p is the proportion of positive class

When evaluating a split at node t using feature j and threshold τ, we compute the **Gini gain**:

$$\Delta G = G(t) - \left(\frac{N_L}{N_t} G(t_L) + \frac{N_R}{N_t} G(t_R) \right)$$

where: - $G(t)$ is the Gini impurity of the parent node - $G(t_L)$ and $G(t_R)$ are the Gini impurities of left and right children - N_t, N_L, N_R are the number of samples in parent, left child, and right child - The split that maximizes ΔG is chosen

0.5 Optimizer: Greedy Recursive Binary Splitting

CART builds trees using a **greedy, top-down, recursive partitioning** algorithm:

1 CART Classifier from Scratch - DATA2060 Final Project

Team: The Overfitters

Authors: Ruoyun Yang, Shiyu Liu, Zhaocheng Yang, Sibo Zhou

GitHub Repository: https://github.com/cynthiayry/DATA2060-Machine-Learning-Algorithm-Project–CART

1.0.1 CART Tree Construction Algorithm

The CART algorithm builds a decision tree recursively using a greedy approach. At each node, it finds the best feature and threshold to split the data by maximizing the Gini gain.

```
Algorithm: BUILD_TREE(D, depth)

Input:  D = {(x_i, y_i)}_{i=1}^N    # Training dataset
        depth                        # Current tree depth
Output: Binary decision tree node


1. Check Stopping Criteria:
   IF any of the following conditions hold:
      • All samples belong to the same class (pure node)
      • depth  max_depth
      • |D| < min_samples_split
   THEN:
      RETURN LEAF_NODE(majority_class(D))

2. Find Best Split:
   Initialize:
      best_gain ← 0
```

```
      best_split ← None

   FOR each feature j  {1, 2, ..., d}:
      # Get unique values of feature j and compute candidate thresholds
      values ← unique_sorted_values(D[:, j])

      FOR each consecutive pair (values[i], values[i+1]):
         # Use midpoint as threshold
          ← (values[i] + values[i+1]) / 2

         # Split dataset based on threshold
         D_left  ← {(x, y)  D : x_j   }
         D_right ← {(x, y)  D : x_j >  }

         # Skip split if it violates min_samples_leaf constraint
         IF |D_left| < min_samples_leaf OR |D_right| < min_samples_leaf:
            CONTINUE

         # Compute Gini gain for this split
         gain ← Gini(D) - (|D_left|/|D|) × Gini(D_left)
                        - (|D_right|/|D|) × Gini(D_right)

         # Update best split if this one is better
         IF gain > best_gain:
            best_gain ← gain
            best_split ← (j,  )

   # If no valid split found, create leaf node
   IF best_gain = 0:
      RETURN LEAF_NODE(majority_class(D))


3. Recursively Build Subtrees:
   (j*,  *) ← best_split
   D_left  ← {(x, y)  D : x_{j*}   *}
   D_right ← {(x, y)  D : x_{j*} >  *}

   node ← INTERNAL_NODE(feature=j*, threshold= *)
   node.left  ← BUILD_TREE(D_left, depth + 1)
   node.right ← BUILD_TREE(D_right, depth + 1)

   RETURN node
```

Key Points: - **Greedy approach**: At each step, we choose the locally optimal split without considering future splits - **Binary splits**: Each node splits on a single feature and threshold, creating exactly two children - **Threshold selection**: We test midpoints between consecutive unique values to find optimal thresholds - **Stopping conditions**: Multiple criteria prevent overfitting and

ensure valid tree structure

Stopping Criteria: - Maximum depth reached (`max_depth`) - Minimum samples required to split (`min_samples_split`) - Minimum samples at leaf node (`min_samples_leaf`) - No improvement in Gini gain - All samples belong to the same class (pure node)

Complexity: - Training: $O(d \cdot N \log N \cdot \text{depth})$ where d is number of features - Prediction: $O(\text{depth})$ which is $O(\log N)$ for balanced trees

1.1 Advantages

1. **Interpretability**: Tree structure is easy to visualize and understand
2. **Non-parametric**: No assumptions about data distribution
3. **Handles mixed data**: Works with both numerical and categorical features
4. **Feature interactions**: Automatically captures feature interactions
5. **Minimal preprocessing**: No need for feature scaling or normalization

1.2 Disadvantages

1. **Overfitting**: Tendency to create overly complex trees that don't generalize
2. **Instability**: Small changes in data can result in very different trees
3. **Bias**: Biased toward features with more levels
4. **Local optimum**: Greedy algorithm doesn't guarantee global optimum
5. **High variance**: Individual trees have high variance (addressed by ensemble methods)

1.3 References

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees*. Wadsworth, Belmont, CA.

Scikit-learn documentation: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegresso

[1]:
```
from __future__ import print_function
from packaging.version import parse as Version
from platform import python_version

OK = '\x1b[42m[ OK ]\x1b[0m'
FAIL = "\x1b[41m[FAIL]\x1b[0m"

try:
    import importlib
except ImportError:
    print(FAIL, "Python version 3.12.11 is required,"
                " but %s is installed." % sys.version)

def import_version(pkg, min_ver, fail_msg=""):
    mod = None
    try:
        mod = importlib.import_module(pkg)
        if pkg in {'PIL'}:
```

```
            ver = mod.VERSION
        else:
            ver = mod.__version__
        if Version(ver) == Version(min_ver):
            print(OK, "%s version %s is installed."
                  % (lib, min_ver))
        else:
            print(FAIL, "%s version %s is required, but %s installed."
                  % (lib, min_ver, ver))
    except ImportError:
        print(FAIL, '%s not installed. %s' % (pkg, fail_msg))
    return mod


# first check the python version
pyversion = Version(python_version())

if pyversion >= Version("3.12.11"):
    print(OK, "Python version is %s" % pyversion)
elif pyversion < Version("3.12.11"):
    print(FAIL, "Python version 3.12.11 is required,"
                " but %s is installed." % pyversion)
else:
    print(FAIL, "Unknown Python version: %s" % pyversion)


print()
requirements = {'matplotlib': "3.10.5", 'numpy': "2.3.2",'sklearn': "1.7.1",
                'pandas': "2.3.2", 'pytest': "8.4.1", 'torch':"2.7.1"}

# now the dependencies
for lib, required_version in list(requirements.items()):
    import_version(lib, required_version)
```

```
[ OK ] Python version is 3.12.11

[ OK ] matplotlib version 3.10.5 is installed.
[ OK ] numpy version 2.3.2 is installed.
[ OK ] sklearn version 1.7.1 is installed.
[ OK ] pandas version 2.3.2 is installed.
[ OK ] pytest version 8.4.1 is installed.
[ OK ] torch version 2.7.1 is installed.
```

1.4 Part 2. Model

[2]:
```
import numpy as np
from collections import Counter

class Node:
    """
    A node in the decision tree.

    Attributes:
        feature_idx (int or None): Index of feature to split on (None for leaf␣
↪nodes)
        threshold (float or None): Threshold value for the split (None for leaf␣
↪nodes)
        left (Node or None): Left child node (samples with feature_idx <=␣
↪threshold)
        right (Node or None): Right child node (samples with feature_idx >␣
↪threshold)
        value (int or None): Class prediction for leaf nodes (None for internal␣
↪nodes)
        gini (float): Gini impurity at this node
        n_samples (int): Number of samples at this node
        class_counts (np.ndarray): Array of counts per class for probability␣
↪estimates
    """
    def __init__(self, feature_idx=None, threshold=None, left=None, right=None,␣
↪value=None, gini=None, n_samples=None, class_counts=None):
        self.feature_idx = feature_idx
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value
        self.gini = gini
        self.n_samples = n_samples
        self.class_counts = class_counts

    def is_leaf(self):
        """
        Check if the node is a leaf node.

        Returns:
            bool: True if the node is a leaf (has a class value), False␣
↪otherwise
        """
        return self.value is not None
```

```python
class DecisionTreeClassifier:
    """
    Decision Tree Classifier using the CART algorithm with Gini impurity.

    This implementation follows the sklearn.tree.DecisionTreeClassifier API
    and uses binary splits with Gini impurity as the splitting criterion.

    Parameters:
        criterion (str): The function to measure split quality. Currently only␣
 ↪'gini' is supported.
        max_depth (int or None): Maximum depth of the tree. If None, nodes␣
 ↪expand until pure or min_samples_split is reached.
        min_samples_split (int): Minimum number of samples required to split an␣
 ↪internal node.
        min_samples_leaf (int): Minimum number of samples required to be at a␣
 ↪leaf node.
        random_state (int or None): Random seed for reproducibility.

    Attributes:
        tree_ (Node): The root node of the fitted tree
        n_features_ (int): Number of features in the training data
        n_classes_ (int): Number of classes in the training data
        classes_ (np.ndarray): Array of class labels
    """

    def __init__(self, criterion='gini', max_depth=None, min_samples_split=2,
                 min_samples_leaf=1, random_state=None):
        self.criterion = criterion
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.random_state = random_state

        self.tree_ = None
        self.n_features_ = None
        self.n_classes_ = None
        self.classes_ = None

        if random_state is not None:
            np.random.seed(random_state)

    def _gini_impurity(self, y):
        """
        Compute Gini impurity for a set of labels.

        Args:
            y (np.ndarray): Array of class labels
```

```
        Returns:
            float: Gini impurity value between 0 (pure) and 1-1/K (maximally␣
↪impure for K classes)
        """
        if len(y) == 0:
            return 0.0
        _, counts = np.unique(y, return_counts=True)
        proportions = counts / len(y)
        return 1.0 - np.sum(proportions ** 2)

    def _split_data(self, X, y, feature_idx, threshold):
        """
        Split dataset into left and right subsets based on a feature and␣
↪threshold.

        Args:
            X (np.ndarray): Feature matrix of shape (n_samples, n_features)
            y (np.ndarray): Target labels of shape (n_samples,)
            feature_idx (int): Index of the feature to split on
            threshold (float): Threshold value for the split

        Returns:
            tuple: (X_left, y_left, X_right, y_right) where left contains␣
↪samples <= threshold
        """
        left_mask = X[:, feature_idx] <= threshold
        right_mask = ~left_mask
        return (X[left_mask], y[left_mask], X[right_mask], y[right_mask])

    def _find_best_split(self, X, y):
        """
        Find the best feature and threshold to split on by maximizing Gini gain.

        Args:
            X (np.ndarray): Feature matrix of shape (n_samples, n_features)
            y (np.ndarray): Target labels of shape (n_samples,)

        Returns:
            tuple: (best_feature_idx, best_threshold, best_gain) or (None,␣
↪None, 0) if no valid split
        """
        n_samples, n_features = X.shape
        if n_samples <= 1:
            return None, None, 0

        parent_gini = self._gini_impurity(y)
```

```
        best_gain, best_feature_idx, best_threshold = 0, None, None

        # Try all features
        for feature_idx in range(n_features):
            feature_values = X[:, feature_idx]
            unique_values = np.unique(feature_values)

            # Try all possible thresholds (midpoints between consecutive unique␣
↪values)
            for i in range(len(unique_values) - 1):
                threshold = (unique_values[i] + unique_values[i + 1]) / 2
                X_left, y_left, X_right, y_right = self._split_data(X, y,␣
↪feature_idx, threshold)

                # Skip if split violates min_samples_leaf constraint
                if len(y_left) < self.min_samples_leaf or len(y_right) < self.
↪min_samples_leaf:
                    continue

                # Compute weighted Gini impurity after split
                n_left, n_right = len(y_left), len(y_right)
                gini_left = self._gini_impurity(y_left)
                gini_right = self._gini_impurity(y_right)
                weighted_gini = (n_left / n_samples) * gini_left + (n_right /␣
↪n_samples) * gini_right

                # Compute Gini gain
                gini_gain = parent_gini - weighted_gini

                if gini_gain > best_gain:
                    best_gain = gini_gain
                    best_feature_idx = feature_idx
                    best_threshold = threshold

        return best_feature_idx, best_threshold, best_gain

    def _build_tree(self, X, y, depth=0):
        """
        Recursively build the decision tree.

        Args:
            X (np.ndarray): Feature matrix of shape (n_samples, n_features)
            y (np.ndarray): Target labels of shape (n_samples,)
            depth (int): Current depth in the tree

        Returns:
            Node: Root node of the constructed (sub)tree
```

```
        """
        n_samples = len(y)
        current_gini = self._gini_impurity(y)

        # Count samples per class
        class_counts_dict = Counter(y)
        class_counts_vec = np.array([class_counts_dict.get(cls, 0) for cls in␣
↪self.classes_])
        majority_class = self.classes_[class_counts_vec.argmax()]

        # Check stopping criteria
        if (
            len(class_counts_dict) == 1 or  # Pure node
            (self.max_depth is not None and depth >= self.max_depth) or  # Max␣
↪depth reached
            n_samples < self.min_samples_split  # Not enough samples to split
        ):
            return Node(value=majority_class, gini=current_gini,␣
↪n_samples=n_samples, class_counts=class_counts_vec)

        # Find best split
        feature_idx, threshold, gini_gain = self._find_best_split(X, y)

        # If no valid split found, create leaf
        if feature_idx is None or gini_gain == 0:
            return Node(value=majority_class, gini=current_gini,␣
↪n_samples=n_samples, class_counts=class_counts_vec)

        # Create split and build children recursively
        X_left, y_left, X_right, y_right = self._split_data(X, y, feature_idx,␣
↪threshold)
        left_child = self._build_tree(X_left, y_left, depth + 1)
        right_child = self._build_tree(X_right, y_right, depth + 1)

        return Node(
            feature_idx=feature_idx,
            threshold=threshold,
            left=left_child,
            right=right_child,
            gini=current_gini,
            n_samples=n_samples,
            class_counts=class_counts_vec
        )

    def fit(self, X, y):
        """
        Fit the decision tree classifier to training data.
```

```
        Args:
            X (array-like): Feature matrix of shape (n_samples, n_features)
            y (array-like): Target labels of shape (n_samples,)

        Returns:
            self: The fitted classifier
        """
        X = np.array(X)
        y = np.array(y)
        self.n_features_ = X.shape[1]
        self.classes_ = np.unique(y)
        self.n_classes_ = len(self.classes_)
        self.tree_ = self._build_tree(X, y)
        return self

    def train(self, X, y):
        """
        Alias for fit() to match homework API convention.

        Args:
            X (array-like): Feature matrix of shape (n_samples, n_features)
            y (array-like): Target labels of shape (n_samples,)

        Returns:
            self: The fitted classifier
        """
        return self.fit(X, y)

    def _predict_sample(self, x, node):
        """
        Predict class label for a single sample by traversing the tree.

        Args:
            x (np.ndarray): Feature vector of shape (n_features,)
            node (Node): Current node in traversal

        Returns:
            int: Predicted class label
        """
        if node.is_leaf():
            return node.value
        if x[node.feature_idx] <= node.threshold:
            return self._predict_sample(x, node.left)
        return self._predict_sample(x, node.right)

    def predict(self, X):
```

```
        """
        Predict class labels for samples in X.

        Args:
            X (array-like): Feature matrix of shape (n_samples, n_features)

        Returns:
            np.ndarray: Predicted class labels of shape (n_samples,)
        """
        X = np.array(X)
        return np.array([self._predict_sample(x, self.tree_) for x in X])

    def _predict_proba_sample(self, x, node):
        """
        Predict class probabilities for a single sample.

        Args:
            x (np.ndarray): Feature vector of shape (n_features,)
            node (Node): Current node in traversal

        Returns:
            np.ndarray: Class probabilities of shape (n_classes,)
        """
        if node.is_leaf():
            counts = node.class_counts
            if counts is None or counts.sum() == 0:
                # Fallback: return one-hot encoding
                proba = np.zeros(self.n_classes_)
                class_idx = np.where(self.classes_ == node.value)[0][0]
                proba[class_idx] = 1.0
                return proba
            return counts / counts.sum()
        if x[node.feature_idx] <= node.threshold:
            return self._predict_proba_sample(x, node.left)
        return self._predict_proba_sample(x, node.right)

    def predict_proba(self, X):
        """
        Predict class probabilities for samples in X.

        Args:
            X (array-like): Feature matrix of shape (n_samples, n_features)

        Returns:
            np.ndarray: Class probabilities of shape (n_samples, n_classes)
        """
        X = np.array(X)
```

```
        return np.array([self._predict_proba_sample(x, self.tree_) for x in X])

    def loss(self, X, y):
        """
        Compute misclassification rate on dataset (X, y).

        Args:
            X (array-like): Feature matrix of shape (n_samples, n_features)
            y (array-like): True labels of shape (n_samples,)

        Returns:
            float: Misclassification rate (fraction of incorrect predictions)
        """
        X = np.array(X)
        y = np.array(y)
        preds = self.predict(X)
        return np.mean(preds != y)

    def score(self, X, y):
        """
        Compute accuracy on dataset (X, y).

        Args:
            X (array-like): Feature matrix of shape (n_samples, n_features)
            y (array-like): True labels of shape (n_samples,)

        Returns:
            float: Accuracy score (fraction of correct predictions)
        """
        return 1.0 - self.loss(X, y)

    def get_depth(self):
        """
        Get the maximum depth of the tree.

        Returns:
            int: Maximum depth of the tree (0 for a tree with only root node)
        """
        def _get_depth(node):
            if node is None or node.is_leaf():
                return 0
            return 1 + max(_get_depth(node.left), _get_depth(node.right))
        return _get_depth(self.tree_)

    def get_n_leaves(self):
        """
        Get the number of leaf nodes in the tree.
```

```
        Returns:
            int: Number of leaf nodes
        """
        def _count_leaves(node):
            if node is None:
                return 0
            if node.is_leaf():
                return 1
            return _count_leaves(node.left) + _count_leaves(node.right)
        return _count_leaves(self.tree_)
```

1.5 Part 3. Check Model

We exercise the CART implementation with method-level unit tests (*gini_impurity*, *split_data*, *find_best_split*, depth/leaf constraints, prediction, probability estimates, loss) and edge cases such as single-class data. We then match sklearn on a synthetic separable dataset and the multiclass Iris benchmark, demonstrating functional parity and deterministic behavior. The final block trains on the Breast Cancer Wisconsin data, reports accuracy, confusion matrix, and depth/leaf counts for both our model and sklearn, and sweeps max_depth to show how capacity impacts bias/variance before saving the plot.

[3]:
```
# Check Model - Unit Tests (HW09-style)

import numpy as np
from pytest import approx
from sklearn.tree import DecisionTreeClassifier as SklearnDTC
from sklearn.datasets import load_iris

np.random.seed(42)

# helper assertions

def check_vals(actual, expected, rtol=1e-6, msg=""):
    assert np.allclose(actual, expected, rtol=rtol), msg


def check_split(X_left, X_right, y_left, y_right, feat_idx, thr):
    assert X_left.shape[0] + X_right.shape[0] == y_left.size + y_right.size,␣
 ↪"split loses samples"
    assert np.all(X_left[:, feat_idx] <= thr), "left branch threshold violation"
    assert np.all(X_right[:, feat_idx] > thr), "right branch threshold␣
 ↪violation"


# Test 1: Gini impurity
clf = DecisionTreeClassifier()
```

```
check_vals(clf._gini_impurity(np.array([1, 1, 1])), 0.0)
check_vals(clf._gini_impurity(np.array([0, 1] * 4)), 0.5)
check_vals(clf._gini_impurity(np.array([0, 0, 1, 1, 2, 2])), 2 / 3)

# Test 2: Data split
X_s = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
y_s = np.array([0, 0, 1, 1])
X_l, y_l, X_r, y_r = clf._split_data(X_s, y_s, feature_idx=0, threshold=4)
check_split(X_l, X_r, y_l, y_r, feat_idx=0, thr=4)

# Test 3: Best split finder (obvious threshold at 3.5)
X_simple = np.array([[1], [2], [3], [4], [5], [6]])
y_simple = np.array([0, 0, 0, 1, 1, 1])
feat, thr, gain = clf._find_best_split(X_simple, y_simple)
assert feat == 0
assert 3 < thr < 4
assert gain > 0.4

# Test 4: Depth constraint respected
X_depth = np.random.randn(100, 3)
y_depth = np.random.randint(0, 2, 100)
for d in [1, 3, 5]:
    model = DecisionTreeClassifier(max_depth=d, random_state=0).fit(X_depth,␣
 ↪y_depth)
    assert model.get_depth() <= d

# Test 5: min_samples_split makes tree shallower
X_mss = np.random.randn(60, 2)
y_mss = np.random.randint(0, 2, 60)
coarse = DecisionTreeClassifier(min_samples_split=20, random_state=0).
 ↪fit(X_mss, y_mss)
fine = DecisionTreeClassifier(random_state=0).fit(X_mss, y_mss)
assert coarse.get_depth() <= fine.get_depth()

# Test 6: min_samples_leaf reduces leaves
X_msl = np.random.randn(120, 2)
y_msl = np.random.randint(0, 2, 120)
wide = DecisionTreeClassifier(min_samples_leaf=10, random_state=0).fit(X_msl,␣
 ↪y_msl)
base = DecisionTreeClassifier(random_state=0).fit(X_msl, y_msl)
assert wide.get_n_leaves() <= base.get_n_leaves()

# Test 7: Predict memorizes small separable set
X_pred = np.array([[1, 1], [5, 5], [2, 2], [6, 6]])
y_pred = np.array([0, 1, 0, 1])
mem = DecisionTreeClassifier(random_state=0).fit(X_pred, y_pred)
check_vals(mem.predict(X_pred), y_pred)
```

```
# Test 8: predict_proba sums to 1 and non-negative
X_prob = np.random.randn(40, 3)
y_prob = np.random.randint(0, 3, 40)
prob_clf = DecisionTreeClassifier(random_state=0).fit(X_prob, y_prob)
probas = prob_clf.predict_proba(X_prob)
check_vals(probas.sum(axis=1), np.ones(probas.shape[0]))
assert np.all((probas >= 0) & (probas <= 1))

# Test 9: score equals manual accuracy
X_sc = np.random.randn(80, 4)
y_sc = np.random.randint(0, 2, 80)
sc_clf = DecisionTreeClassifier(max_depth=4, random_state=0).fit(X_sc, y_sc)
acc_manual = np.mean(sc_clf.predict(X_sc) == y_sc)
check_vals(sc_clf.score(X_sc, y_sc), acc_manual)

# Test 10: Single-class dataset -> depth 0, one leaf, constant preds
X_one = np.random.randn(25, 3)
y_one = np.ones(25, dtype=int)
one_clf = DecisionTreeClassifier().fit(X_one, y_one)
assert one_clf.get_depth() == 0
assert one_clf.get_n_leaves() == 1
check_vals(one_clf.predict(X_one), y_one)

# Test 11: Sklearn parity on simple synthetic data
X_syn = np.random.randn(50, 3)
y_syn = (X_syn[:, 0] + X_syn[:, 1] > 0).astype(int)
ours_syn = DecisionTreeClassifier(max_depth=3, min_samples_split=5,␣
 ↪random_state=0).fit(X_syn, y_syn)
sk_syn = SklearnDTC(max_depth=3, min_samples_split=5, random_state=0).
 ↪fit(X_syn, y_syn)
check_vals(ours_syn.score(X_syn, y_syn), sk_syn.score(X_syn, y_syn), rtol=0.15)

# Test 12: Sklearn parity on Iris
iris = load_iris()
X_iris, y_iris = iris.data, iris.target
ours_iris = DecisionTreeClassifier(max_depth=4, random_state=0).fit(X_iris,␣
 ↪y_iris)
sk_iris = SklearnDTC(max_depth=4, random_state=0).fit(X_iris, y_iris)
check_vals(ours_iris.score(X_iris, y_iris), sk_iris.score(X_iris, y_iris),␣
 ↪rtol=0.10)

# Test 13: Exact reproduction on controlled toy data
X_exact = np.array([[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5]])
y_exact = np.array([0, 0, 0, 1, 1, 1])
ours_exact = DecisionTreeClassifier(max_depth=2, random_state=0).fit(X_exact,␣
 ↪y_exact)
```

```
sk_exact = SklearnDTC(max_depth=2, random_state=0).fit(X_exact, y_exact)
assert np.array_equal(ours_exact.predict(X_exact), sk_exact.predict(X_exact))

# Test 14: loss equals misclassification rate
X_loss = np.array([[0], [1], [2], [3]])
y_loss = np.array([0, 0, 1, 1])
loss_clf = DecisionTreeClassifier(max_depth=2, random_state=0).fit(X_loss,␣
 ↪y_loss)
assert loss_clf.loss(X_loss, y_loss) == approx(0.0, abs=1e-6)

print("unit tests completed (HW-style checks)")
```

```
unit tests completed (HW-style checks)
```

1.6 Part 4. Main

End-to-end experiment on the Breast Cancer Wisconsin dataset (Dua and Graff, 2019; Street et al., 1993): load the provided train/validation splits, fit our CART with sensible depth/splitting constraints, compare side-by-side with `sklearn.tree.DecisionTreeClassifier`, report accuracy and a confusion matrix, and sweep `max_depth` to visualize the bias–variance trade-off.

[4]:
```
# Main - Run CART on Breast Cancer Dataset

import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier as SklearnDTC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
import matplotlib.pyplot as plt

def load_breast_cancer_data():
    """
    Load and prepare the Breast Cancer Wisconsin dataset.
    This assumes you have the data in the same format as HW04.

    If you don't have the exact files, you can use sklearn's built-in dataset:
    from sklearn.datasets import load_breast_cancer
    data = load_breast_cancer()
    X, y = data.data, data.target
    """
    try:
        # Try loading from data directory (HW04 format)
        X_train = pd.read_csv('data/X_train.csv', header=None)
        Y_train = pd.read_csv('data/y_train.csv', header=None)
        X_val = pd.read_csv('data/X_val.csv', header=None)
        Y_val = pd.read_csv('data/y_val.csv', header=None)
```

```
        Y_train = np.array([i[0] for i in Y_train.values])
        Y_val = np.array([i[0] for i in Y_val.values])

        X_train = np.array(X_train)
        X_val = np.array(X_val)

        print("Loaded data from data/ directory")
        return X_train, X_val, Y_train, Y_val

    except FileNotFoundError:
        # Fall back to sklearn's built-in dataset
        print("data/ directory not found, using sklearn's built-in breast␣
 ↪cancer dataset")
        from sklearn.datasets import load_breast_cancer

        data = load_breast_cancer()
        X, y = data.data, data.target

        # Split into train and validation sets (80-20 split)
        X_train, X_val, Y_train, Y_val = train_test_split(
            X, y, test_size=0.2, random_state=42, stratify=y
        )

        return X_train, X_val, Y_train, Y_val


def main():
    print("=" * 70)
    print("CART DECISION TREE CLASSIFIER - BREAST CANCER CLASSIFICATION")
    print("=" * 70)

    # Load data
    print("\n### Loading Data ###")
    X_train, X_val, Y_train, Y_val = load_breast_cancer_data()

    print(f"Training set size: {X_train.shape[0]} samples, {X_train.shape[1]}␣
 ↪features")
    print(f"Validation set size: {X_val.shape[0]} samples")
    print(f"Class distribution in training: {np.bincount(Y_train)}")

    # Train our CART implementation
    print("\n### Training Our CART Implementation ###")
    our_clf = DecisionTreeClassifier(
        max_depth=5,
        min_samples_split=10,
        min_samples_leaf=5,
        random_state=42
```

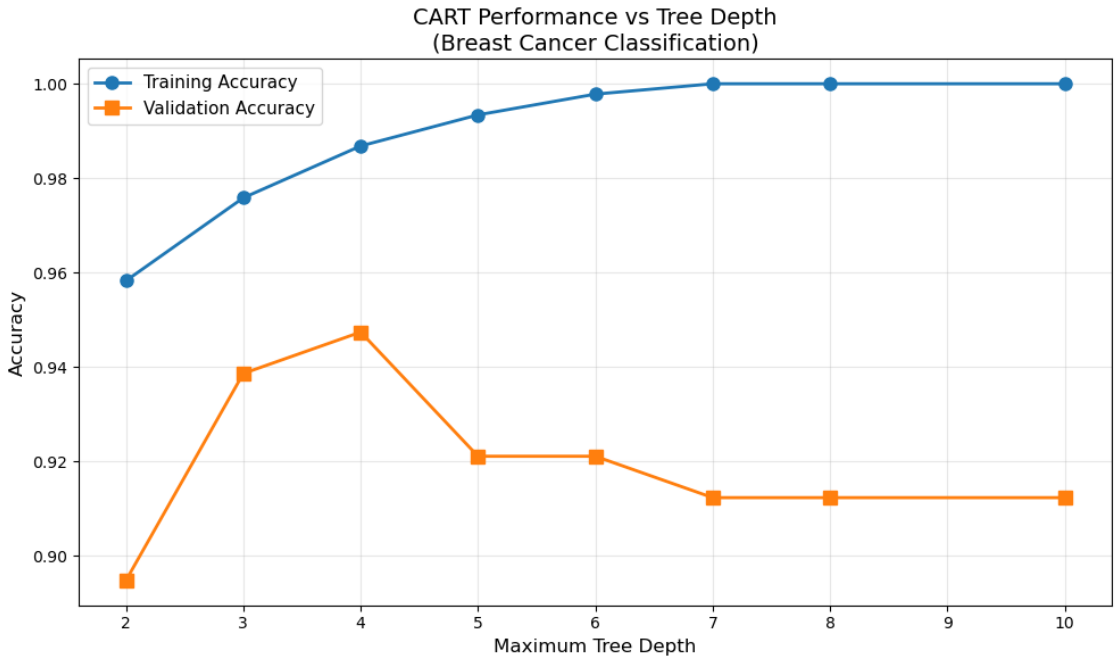
```
    )
    our_clf.fit(X_train, Y_train)

    # Make predictions
    train_pred = our_clf.predict(X_train)
    val_pred = our_clf.predict(X_val)

    # Calculate accuracies
    train_acc = our_clf.score(X_train, Y_train)
    val_acc = our_clf.score(X_val, Y_val)

    print(f"\nOur Implementation Results:")
    print(f"  Training Accuracy: {train_acc:.4f}")
    print(f"  Validation Accuracy: {val_acc:.4f}")
    print(f"  Tree Depth: {our_clf.get_depth()}")
    print(f"  Number of Leaves: {our_clf.get_n_leaves()}")

    # Train sklearn's implementation for comparison
    print("\n### Training sklearn's Implementation ###")
    sk_clf = SklearnDTC(
        max_depth=5,
        min_samples_split=10,
        min_samples_leaf=5,
        random_state=42
    )
    sk_clf.fit(X_train, Y_train)

    sk_train_pred = sk_clf.predict(X_train)
    sk_val_pred = sk_clf.predict(X_val)

    sk_train_acc = sk_clf.score(X_train, Y_train)
    sk_val_acc = sk_clf.score(X_val, Y_val)

    print(f"\nsklearn Implementation Results:")
    print(f"  Training Accuracy: {sk_train_acc:.4f}")
    print(f"  Validation Accuracy: {sk_val_acc:.4f}")
    print(f"  Tree Depth: {sk_clf.get_depth()}")
    print(f"  Number of Leaves: {sk_clf.get_n_leaves()}")

    # Comparison
    print("\n### Comparison ###")
    print(f"Training Accuracy Difference: {abs(train_acc - sk_train_acc):.6f}")
    print(f"Validation Accuracy Difference: {abs(val_acc - sk_val_acc):.6f}")
    print(f"Tree Depth Difference: {abs(our_clf.get_depth() - sk_clf.
↪get_depth())}")

    # Detailed classification report
```

```
    print("\n### Detailed Classification Report (Our Implementation) ###")
    print(classification_report(Y_val, val_pred, target_names=['Malignant',␣
↪'Benign']))

    # Confusion matrix
    print("\n### Confusion Matrix (Our Implementation) ###")
    cm = confusion_matrix(Y_val, val_pred)
    print(cm)
    print("\nConfusion Matrix Interpretation:")
    print(f"  True Negatives (Malignant correctly classified): {cm[0, 0]}")
    print(f"  False Positives (Malignant misclassified as Benign): {cm[0, 1]}")
    print(f"  False Negatives (Benign misclassified as Malignant): {cm[1, 0]}")
    print(f"  True Positives (Benign correctly classified): {cm[1, 1]}")

    # Hyperparameter tuning demonstration
    print("\n### Hyperparameter Tuning ###")
    print("Testing different max_depth values...")

    depths = [2, 3, 4, 5, 6, 7, 8, 10]
    train_accs = []
    val_accs = []

    for depth in depths:
        clf_temp = DecisionTreeClassifier(max_depth=depth, random_state=42)
        clf_temp.fit(X_train, Y_train)
        train_accs.append(clf_temp.score(X_train, Y_train))
        val_accs.append(clf_temp.score(X_val, Y_val))
        print(f"  max_depth={depth:2d}: Train Acc={train_accs[-1]:.4f}, Val␣
↪Acc={val_accs[-1]:.4f}")

    # Plot accuracy vs depth
    plt.figure(figsize=(10, 6))
    plt.plot(depths, train_accs, 'o-', label='Training Accuracy', linewidth=2,␣
↪markersize=8)
    plt.plot(depths, val_accs, 's-', label='Validation Accuracy', linewidth=2,␣
↪markersize=8)
    plt.xlabel('Maximum Tree Depth', fontsize=12)
    plt.ylabel('Accuracy', fontsize=12)
    plt.title('CART Performance vs Tree Depth\n(Breast Cancer Classification)',␣
↪fontsize=14)
    plt.legend(fontsize=11)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig('cart_depth_analysis.png', dpi=150, bbox_inches='tight')
    print("\nSaved plot as 'cart_depth_analysis.png'")
    plt.show()
```

```
    # Find best depth
    best_depth_idx = np.argmax(val_accs)
    best_depth = depths[best_depth_idx]
    best_val_acc = val_accs[best_depth_idx]
    print(f"\nBest max_depth: {best_depth} (Validation Accuracy: {best_val_acc:.
 ↪4f})")

    print("=" * 70)

# Set random seed for reproducibility
np.random.seed(42)

# Run main
main()
```

```
======================================================================
CART DECISION TREE CLASSIFIER - BREAST CANCER CLASSIFICATION
======================================================================

### Loading Data ###
data/ directory not found, using sklearn's built-in breast cancer dataset
Training set size: 455 samples, 30 features
Validation set size: 114 samples
Class distribution in training: [170 285]

### Training Our CART Implementation ###

Our Implementation Results:
  Training Accuracy: 0.9758
  Validation Accuracy: 0.9298
  Tree Depth: 5
  Number of Leaves: 11

### Training sklearn's Implementation ###

sklearn Implementation Results:
  Training Accuracy: 0.9758
  Validation Accuracy: 0.9211
  Tree Depth: 5
  Number of Leaves: 12

### Comparison ###
Training Accuracy Difference: 0.000000
Validation Accuracy Difference: 0.008772
Tree Depth Difference: 0

### Detailed Classification Report (Our Implementation) ###
              precision    recall  f1-score   support
```

```
   Malignant       0.90      0.90      0.90        42
      Benign       0.94      0.94      0.94        72

    accuracy                           0.93       114
   macro avg       0.92      0.92      0.92       114
weighted avg       0.93      0.93      0.93       114


### Confusion Matrix (Our Implementation) ###
[[38  4]
 [ 4 68]]

Confusion Matrix Interpretation:
  True Negatives (Malignant correctly classified): 38
  False Positives (Malignant misclassified as Benign): 4
  False Negatives (Benign misclassified as Malignant): 4
  True Positives (Benign correctly classified): 68

### Hyperparameter Tuning ###
Testing different max_depth values…
  max_depth= 2: Train Acc=0.9582, Val Acc=0.8947
  max_depth= 3: Train Acc=0.9758, Val Acc=0.9386
  max_depth= 4: Train Acc=0.9868, Val Acc=0.9474
  max_depth= 5: Train Acc=0.9934, Val Acc=0.9211
  max_depth= 6: Train Acc=0.9978, Val Acc=0.9211
  max_depth= 7: Train Acc=1.0000, Val Acc=0.9123
  max_depth= 8: Train Acc=1.0000, Val Acc=0.9123
  max_depth=10: Train Acc=1.0000, Val Acc=0.9123

Saved plot as 'cart_depth_analysis.png'
```




```
Best max_depth: 4 (Validation Accuracy: 0.9474)
======================================================================
```

1.7 References

- Breiman, L., Friedman, J.H., Olshen, R.A. and Stone, C.J. (1984) *Classification and Regression Trees*. Wadsworth, Belmont, CA.
- Loh, W.Y. (2011) 'Classification and regression trees', *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1), pp. 14–23.
- Pedregosa, F. *et al.* (2011) 'Scikit-learn: Machine learning in Python', *Journal of Machine Learning Research*, 12, pp. 2825–2830.
- Dua, D. and Graff, C. (2019) 'UCI Machine Learning Repository'. University of California, Irvine, School of Information and Computer Science. Available at: https://archive.ics.uci.edu/ml (Accessed: 6 December 2025).
- Street, W.N., Wolberg, W.H. and Mangasarian, O.L. (1993) 'Nuclear feature extraction for breast tumor diagnosis', *Biomedical Image Processing and Biomedical Visualization*, pp. 861–870.