DATA2060 Final Project

Model: **CART for classification**
Github repo: https://github.com/mindyxu0125/Data2060_Human_not_learning.git

Team Members:

- Muxin Fu
- Yixiao Zhang
- Jingmin Xu
- Mingrui Chen

Part 1: Overview of CART for Classification

0. Introduction

0.1 Overview

The Classification and Regression Tree (CART) algorithm is a nonparametric supervised learning method that builds a binary decision tree for classification tasks.

At each step, the algorithm selects a feature and threshold that create two child nodes with lower class impurity, using criteria such as Gini impurity or entropy. Through this recursive partitioning, CART represents the classifier as a set of piecewise-constant regions, where each leaf corresponds to a predicted class label. Because the sequence of splits directly mirrors the decision-making process, CART offers a transparent and intuitive model structure.

0.2 Advantages

CART offers several notable strengths that contribute to its widespread use as a baseline classifier.

- The key difference is that CART opens the window for allowing **splits on continunous features**, where it applies the threshold-splitting. Therefore, the model is highly interpretable: each internal node corresponds to a clear "if–then" condition based on a single feature, allowing the entire decision path to be easily traced and communicated. This transparency is particularly valuable in settings where model explanations are required.

- Second, CART is able to **capture nonlinear relationships** and feature interactions without relying on explicit transformations or parametric assumptions. Its recursive splitting procedure enables the model to adapt flexibly to irregular or complex decision boundaries, providing expressive power beyond that of linear models.

- Moreover, CART requires **minimal preprocessing**. It can accommodate both numerical and categorical variables, is robust to monotonic feature scaling, and implicitly performs feature selection by choosing splits only on informative variables. These characteristics make CART convenient to implement and reliable across a wide range of practical applications.

0.3 Disadvantages

Despite its advantages, CART also presents several limitations that must be considered.

- Most importantly, the model is prone to overfitting when allowed to grow without constraints. As emphasized in the bias–complexity trade-off discussed in the course reading, increasing model flexibility reduces approximation error but raises estimation error, causing deep, unpruned trees to exhibit high variance and poor generalization.

- CART also tends to be unstable: small perturbations in the training data can alter early splits, resulting in substantially different tree structures. This sensitivity undermines the model's reliability, especially in contexts requiring stable predictions.
- Finally, because CART relies exclusively on axis-aligned splits, it may need many successive partitions to approximate diagonal or curved decision boundaries, leading to unnecessarily deep and complex trees. These shortcomings motivate the use of pruning techniques and more advanced ensemble methods, such as Random Forests and Gradient Boosting, which address variance and stability issues more effectively.

1. Representation

1.1 **Domain Set**

We define the domain space as

In the CART classification setting, each training example is represented as a feature vector in an n-dimensional real space:

$$\mathcal{X} = \mathbb{R}^n, \qquad x_i = (x_{i1}, x_{i2}, \dots, x_{in}) \in \mathcal{X}.$$

Each component x_{ij} represents the value of feature j for sample i. The feature domain can include continuous or categorical variables (encoded numerically in practice).

1.2 **Label Set**

For a K-class classification task, the label space is defined as:

$$\mathcal{Y} = \{0, 1, \dots, K-1\}.$$

In the binary case, this simplifies to:

$$\mathcal{Y} = \{0, 1\}.$$

1.3 **Training Data**

We are given a labeled dataset:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}, \quad x_i \in \mathcal{X},\ y_i \in \mathcal{Y}.$$

Each pair (x_i, y_i) represents one training example. The training process recursively partitions $\mathcal{D}$ based on feature thresholds to form a binary decision tree.

1.4 **Learner's Output**

Formally, the hypothesis space of CART classification is defined as the set of **binary decision trees** of depth at most $T_{\max}$:

$$\mathcal{H} = \{\, h : \mathcal{X} \to \mathcal{Y} \mid h \text{ is a binary decision tree with depth } \le T_{\max} \,\}.$$

Each decision tree $h \in \mathcal{H}$ recursively partitions the input space $\mathcal{X}$ into at most $2^{T_{\max}}$ disjoint leaves.

At prediction time, a new observation x is passed through the sequence of feature tests $(x_f \le t)$ until it reaches a leaf node i. Each leaf stores an empirical class probability vector

$$p_i = (p_{i,0}, p_{i,1}, \dots, p_{i,K-1}),$$

computed from the training samples that reached that leaf.

The predicted class label is then determined by

$$\hat{y}(x) = \arg\max_k \, p_{i,k}.$$

2. Loss

In the classification setting, losses are the **measures of impurity**. CART minimizes impruity and the loss is defined per split. Generally speaking, **Gini** and **Entropy** are good measures.

To compute Loss, we need:

- Impurity measure,
- Split loss based on chosen impurity measure.

In the scikit-learn, this is determined by the parameter **criterion**: *{"gini", "entropy", "log_loss"}, default="gini"*

2.1 **Impurity Function**

For a K-class classification problem, consider node i containing a subset of samples

$$S_i = \{(x_j, y_j)\}_{j \in \mathcal{I}_i}, \qquad N_i = |S_i|.$$

The number of samples in node i that belong to class k is

$$n_{i,k} = \sum_{j \in \mathcal{I}_i} \mathbf{1}(y_j = k).$$

The class proportion of class k in node i is

$$p_{i,k} = \frac{n_{i,k}}{N_i}, \qquad k = 1, \dots, K.$$

$$\sum_{k=1}^{K} p_{i,k} = 1, \text{and } p_{i,k} \geq 0 \quad \text{for } k = 1, \dots, K.$$

2.1.1 **Gini**

- The Gini impurity of node i is:

$$G_i = 1 - \sum_{k=1}^{K} p_{i,k}^2.$$

2.1.2 **Entropy**

- The entropy impurity of node i is

$$H_i = -\sum_{k=1}^{K} p_{i,k} \log p_{i,k},$$

- And we assume $0 \log 0 = 0$.

2.2 **Split Loss**

Given a candidate split θ applied at node i, the dataset S_i is partitioned into a left subset $S_i^{\text{left}}(\theta)$ and a right subset $S_i^{\text{right}}(\theta)$:

$$S_i^{\text{left}}(\theta) = \{(x_j, y_j) \in S_i \mid x_{j,f} \le t\},$$

$$S_i^{\text{right}}(\theta) = S_i \setminus S_i^{\text{left}}(\theta),$$

where $\theta = (f, t)$ denotes the split feature index f and the threshold value t.

Let the number of samples in the left and right subsets be

$$N_i^{\text{left}} = |S_i^{\text{left}}(\theta)|, \qquad N_i^{\text{right}} = |S_i^{\text{right}}(\theta)|.$$

Their corresponding class proportions are computed in the same way as in Section 2.1.

2.2.1 **Weighted Child Impurity**

Given an impurity function $C(\cdot)$ (e.g., Gini or entropy), the **split loss** at node i for candidate split θ is defined as the weighted sum of the left and right child impurities:

$$L(S_i, \theta) = \frac{N_i^{\text{left}}}{N_i} C\big(S_i^{\text{left}}(\theta)\big) \; + \; \frac{N_i^{\text{right}}}{N_i} C\Big(S_i^{\text{right}}(\theta)\Big) \, .$$

Here:

- $C\big(S_i^{\text{left}}(\theta)\big)$ is the impurity (Gini or entropy) of the left child node.
- $C\Big(S_i^{\text{right}}(\theta)\Big)$ is the impurity of the right child node.

2.2.2 **Optimal Split Selection**

The optimal split parameter is chosen by minimizing the split loss:

$$\theta^* = \arg\min_{\theta} \; L(S_i, \theta).$$

And this will be futher explained in the next part, Optimizer on how to actually implement it.

3. Optimizer

3.1 What is Optimized in CART

CART performs a **greedy, recursive partitioning** - at each node, it selects the best split that maximizes information gain (or equivalently minimizes impurity).

So the optimizer is essentially a **greedy search algorithm** that finds:

$$\arg\min_{(f,t)} \; \text{Impurity}(S_{\text{left}}) + \text{Impurity}(S_{\text{right}})$$

where f is the feature and t is the threshold.

3.1.1 Objective Function

CART minimizes an **impurity measure** (loss function) such as:

- Gini Index:

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

- Entropy:

$$H(S) = -\sum_{k=1}^{K} p_k log(p_k)$$

At each node:

$$\mathrm{Gain}(S, f, t) = \mathrm{Impurity}(S) - \frac{|S_{\mathrm{left}}|}{|S|}\mathrm{Impurity}(S_{\mathrm{left}}) - \frac{|S_{\mathrm{right}}|}{|S|}\mathrm{Impurity}(S_{\mathrm{right}})$$

The algorithm chooses the feature $f*$ and threshold $t*$ that maximize this gain.

3.1.2 Pseudo-code

Intuitively, the algorithm asks: “Which feature and cutoff most cleanly separates the classes?” By evaluating all possible splits and picking the one that reduces impurity the most, CART greedily chooses the single question that best organizes the data at this point in the tree. This local optimization step is repeated recursively to grow the whole decision tree.

In below, we have two parts of pseudo-code, explain how CART 1) build the tree recursively, and 2) optimize for the best split at each split.

Pseudo-Code: Tree Construction (with Explicit Stopping Criteria)

Goal: Recursively grow the CART tree using greedy splits and well-defined stopping rules.

```
FUNCTION BUILD_TREE(S_i, depth):

    imp ← impurity(S_i)
    n_samples ← number of samples in S_i

    1. IF imp == 0:
          # Node is pure — all samples belong to the same class
          RETURN LeafNode(class_distribution(S_i))

    2. IF n_samples < min_samples_split:
          # Too few samples to reliably split
          RETURN LeafNode(class_distribution(S_i))

    3. IF depth ≥ max_depth:
          # Depth limit reached
          RETURN LeafNode(class_distribution(S_i))

    4. IF no_valid_threshold_exists(S_i):
          # All features have only one unique value, or all splits produce empty child nodes
          RETURN LeafNode(class_distribution(S_i))
    (j*, t*) ← FIND_BEST_SPLIT(S_i)

    IF j* is None:
        RETURN LeafNode(class_distribution(S_i))

    S_left  ← samples in S_i where x_j* ≤ t*
    S_right ← samples in S_i where x_j* >  t*

    LeftChild  ← BUILD_TREE(S_left,  depth + 1)
    RightChild ← BUILD_TREE(S_right, depth + 1)

    RETURN InternalNode(
                feature = j*,
                threshold = t*,
                left = LeftChild,
                right = RightChild
           )
```

Pseudo-Code: Best Split Search

Goal: find the best feature-threshold pair `(j*, t*)` minimizing node loss.

```
FUNCTION FIND_BEST_SPLIT(S_i):

    best_feature ← None
    best_threshold ← None
    best_loss ← +∞

    FOR each feature j in 1..d:

        x_j ← column j of S_i
        v ← sorted unique values of x_j

        IF length(v) == 1:
            CONTINUE  // no valid split

        T_j ← midpoints of consecutive values in v

        FOR each t in T_j:

            S_left  ← samples with x_j ≤ t
            S_right ← samples with x_j > t

            IF S_left empty OR S_right empty:
                CONTINUE

            imp_left  ← impurity(S_left)
            imp_right ← impurity(S_right)

            loss ← (|S_left|/|S_i|)*imp_left + (|S_right|/|S_i|)*imp_right

            IF loss < best_loss:
                best_loss ← loss
                best_feature ← j
                best_threshold ← t

    RETURN (best_feature, best_threshold)
```

Pseudo-Code: Cost-Complexity Pruning Path

Goal: find the pruning path.

```
FUNCTION COST_COMPLEXITY_PRUNING_PATH(tree):

    # Step 1 — Compute subtree impurity and leaf count (bottom-up)
    FOR each node t (post-order traversal):
        IF t is a leaf:
            R_subtree[t] ← impurity(t) * sample_count(t)
            num_leaves[t] ← 1
        ELSE:
            R_subtree[t] ← R_subtree[left_child(t)] + R_subtree[right_child(t)]
            num_leaves[t] ← num_leaves[left_child(t)] + num_leaves[right_child(t)]

    # Step 2 — Compute the "cost of pruning" αₜ for each internal node
    FOR each internal node t:
        R_leaf ← impurity( t treated as a leaf ) * sample_count(t)
        α_t ← ( R_leaf − R_subtree[t] ) / ( num_leaves[t] − 1 )

    # Step 3 — Iteratively prune subtrees with the smallest αₜ
    α_list ← [0]
    impurity_list ← [ R_subtree[root] ]

    WHILE the tree can still be pruned:
        α_min ← minimum α_t over all remaining internal nodes
        prune all subtrees whose α_t = α_min (replace with a leaf)
        update the total tree impurity
        append α_min to α_list
        append updated impurity to impurity_list

    RETURN (α_list, impurity_list)
```

Part 2: Model

In below is our code for model class, tree class, impurity functions, and other helper functions.

In [ ]:
```python
import numpy as np

def node_score_gini_from_counts(counts):
    '''
    Compute Gini impurity from class counts directly.
    '''
    n = int(counts.sum())
    if n == 0:
        return 0.0
    sum_sq = float((counts ** 2).sum())
    return 1.0 - sum_sq / (n * n)

def node_score_entropy_from_counts(counts):
    '''
    Compute Entropy impurity from class counts directly:
        H = log2(n) - (1/n) * sum_k c_k * log2(c_k)
    '''
    n = int(counts.sum())
    if n == 0:
        return 0.0

    mask = counts > 0
    if not np.any(mask):
        return 0.0

    c = counts[mask]
    return float(np.log2(n) - (c * np.log2(c)).sum() / n)

## Tree Structure to mimic sklearn's DecisionTreeClassifier tree storage
class _Tree:
    """
    Simple tree structure storing node info in parallel lists, then
    converted to numpy arrays via finalize().
    """

    def __init__(self, n_classes):
        self.n_classes = n_classes

        self.children_left = []
        self.children_right = []
        self.feature = []
        self.threshold = []
        self.impurity = []
        self.n_node_samples = []
        self.value = []   # class counts per node (1D arrays length n_classes)

    def add_node(self, feature, threshold, impurity,
                 n_node_samples, counts, left=-1, right=-1):
        """
        Append a node and return its node_id (index).
        feature = -1 means leaf.
        """
        node_id = len(self.feature)
        self.children_left.append(int(left))
        self.children_right.append(int(right))
        self.feature.append(int(feature))
        self.threshold.append(float(threshold))
        self.impurity.append(float(impurity))
        self.n_node_samples.append(int(n_node_samples))
        self.value.append(np.asarray(counts, dtype=np.int64))
        return node_id

    def finalize(self):
        """
        Convert internal Python lists to numpy arrays.
        """
        self.children_left = np.asarray(self.children_left, dtype=np.int32)
        self.children_right = np.asarray(self.children_right, dtype=np.int32)
        self.feature = np.asarray(self.feature, dtype=np.int32)
        self.threshold = np.asarray(self.threshold, dtype=np.float64)
        self.impurity = np.asarray(self.impurity, dtype=np.float64)
        self.n_node_samples = np.asarray(self.n_node_samples, dtype=np.int64)
        self.value = np.stack(self.value, axis=0)  # shape (n_nodes, n_classes)
        self.node_count = self.feature.shape[0]




class CARTClassifier:
    """
    Numpy-only CART decision tree classifier with:
    - criterion: "gini" or "entropy"
    - max_depth: maximum depth of the tree (or None)
    - min_sample_split: minimum samples required to split
    - alpha: cost-complexity pruning parameter (0 = no pruning)
    - random_state: used ONLY to randomly permute feature order at each split
    """

    def __init__(self,
                 criterion="gini",
                 max_depth=None,
                 min_sample_split=2,
                 alpha=0.0,
                 random_state=None):

        self.criterion = criterion
        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.alpha = float(alpha)
        self.random_state = random_state
        self._rng = np.random.RandomState(random_state)
        if criterion == "gini":
            self._impurity_from_counts = node_score_gini_from_counts
        else:
            self._impurity_from_counts = node_score_entropy_from_counts

    def fit(self, X, y):
        """
        Build the full tree, then apply cost-complexity pruning with alpha.
        """
        X = np.asarray(X, dtype=np.float64)
        y = np.asarray(y, dtype=np.int64)

        self.n_samples_, self.n_features_in_ = X.shape

        # Handle class labels (0..K-1 or need remap)
```

```python
        classes = np.unique(y)
        if not np.array_equal(classes, np.arange(classes.size)):
            # remap to 0..K-1
            self._class_mapping_ = {c: i for i, c in enumerate(classes)}
            y_enc = np.array([self._class_mapping_[c] for c in y], dtype=np.int64)
            self.classes_ = classes
        else:
            self._class_mapping_ = None
            y_enc = y
            self.classes_ = classes

        self.n_classes_ = self.classes_.size

        # Build full (unpruned) tree
        self.tree_ = _Tree(n_classes=self.n_classes_)
        indices = np.arange(self.n_samples_, dtype=np.int64)
        self._build_tree(X, y_enc, indices, depth=0)
        self.tree_.finalize()

        # Cost-complexity pruning with given alpha
        if self.alpha > 0.0:
            self._prune_tree()

        return self

    # Internal helpers
    def _class_counts(self, y_subset):
        return np.bincount(y_subset, minlength=self.n_classes_)

    def _build_tree(self, X, y, indices, depth):
        """
        Recursively build the tree using greedy splitting.
        Returns node_id of the root of this subtree.
        """
        y_node = y[indices]
        counts = self._class_counts(y_node)
        n_node_samples = indices.size
        impurity = self._impurity_from_counts(counts)

        # Stopping criteria
        # 1) Pure node (allow tiny negative -0.0 from fp)
        if impurity <= 0.0:
            return self.tree_.add_node(feature=-1,threshold=-1.0,
                                       impurity=impurity,n_node_samples=n_node_samples,
                                       counts=counts,left=-1,right=-1)

        # 2) Too few samples
        if n_node_samples < self.min_sample_split:
            return self.tree_.add_node(feature=-1,threshold=-1.0,
                                       impurity=impurity,n_node_samples=n_node_samples,
                                       counts=counts,left=-1,right=-1)

        # 3) Depth limit
        if self.max_depth is not None and depth >= self.max_depth:
            return self.tree_.add_node(feature=-1,threshold=-1.0,
                                       impurity=impurity,n_node_samples=n_node_samples,
                                       counts=counts,left=-1,right=-1)

        # Find best split
        best_feature, best_threshold, best_loss = self._find_best_split(X, y_node, indices)
        # 4) No valid split found -> leaf
        if best_feature is None:
            return self.tree_.add_node(feature=-1,threshold=-1.0,impurity=impurity,
                                       n_node_samples=n_node_samples,
                                       counts=counts,left=-1,right=-1)

        # Partition samples
        x_best = X[indices, best_feature]
        left_mask = x_best <= best_threshold
        right_mask = ~left_mask

        # Safety: if split degenerate, fallback to leaf
        if (not np.any(left_mask)) or (not np.any(right_mask)):
            return self.tree_.add_node(feature=-1,threshold=-1.0,impurity=impurity,
                                       n_node_samples=n_node_samples,
                                       counts=counts,left=-1,right=-1)

        idx_left = indices[left_mask]
        idx_right = indices[right_mask]

        # Create internal node (children set after recursion)
        node_id = self.tree_.add_node(feature=int(best_feature),threshold=float(best_threshold),
                                      impurity=float(impurity),n_node_samples=int(n_node_samples),
                                      counts=counts,left=-1,right=-1)

        # Recursively build children
        left_child = self._build_tree(X, y, idx_left, depth + 1)
        right_child = self._build_tree(X, y, idx_right, depth + 1)

        # Patch children pointers
        self.tree_.children_left[node_id] = left_child
        self.tree_.children_right[node_id] = right_child

        return node_id

    def _find_best_split(self, X, y_node, indices):
        """
        Find best (feature, threshold) for the node defined by `indices`.
        """
        best_feature = None
        best_threshold = None
        best_loss = np.inf
        EPS = 1e-12

        # per-node random permutation of features
        feature_indices = self._rng.permutation(self.n_features_in_)

        y_sub = y_node

        for j in feature_indices:
            x_j = X[indices, j]
            # Sort by feature value
            order = np.argsort(x_j, kind="mergesort")
            x_sorted = x_j[order]
            y_sorted = y_sub[order]

            # No split if all values equal
            if x_sorted[0] == x_sorted[-1]:
                continue
            # Candidate split positions: k where x[k] != x[k+1]
            diff = x_sorted[1:] != x_sorted[:-1]
            if not np.any(diff):
                continue
            split_pos = np.nonzero(diff)[0]  # array of k indices

            # Initialize class counts
            right_counts = self._class_counts(y_sorted)  # all samples start on right
            left_counts = np.zeros(self.n_classes_, dtype=np.int64)

            # We'll sweep once from left->right, and only evaluate at split_pos.
            sp_i = 0
            next_k = split_pos[sp_i]

            # Move sample k from right to left each step
            for k in range(x_sorted.size - 1):
                cls = y_sorted[k]
                left_counts[cls] += 1
                right_counts[cls] -= 1

                if k != next_k:
                    continue

                nL = k + 1
                nR = x_sorted.size - nL
                if nL == 0 or nR == 0:
                    pass
                else:
                    imp_left = self._impurity_from_counts(left_counts)
                    imp_right = self._impurity_from_counts(right_counts)
                    loss = nL * imp_left + nR * imp_right

                    # STRICT improvement only => first best encountered wins
                    if loss < best_loss - EPS:
                        best_loss = loss
                        best_feature = j
                        best_threshold = 0.5 * (x_sorted[k] + x_sorted[k + 1])
                sp_i += 1
                if sp_i >= split_pos.size:
                    break
                next_k = split_pos[sp_i]
        return best_feature, best_threshold, best_loss

    # Cost-complexity pruning
    def _compute_subtree_stats(self, node_id):
        """
        Compute:
            R_subtree = sum impurity(leaf) * n_samples(leaf)
            n_leaves  = number of leaves
        for the subtree rooted at node_id.
        """
        left = self.tree_.children_left[node_id]
        right = self.tree_.children_right[node_id]

        if left == -1 and right == -1:
            R = self.tree_.impurity[node_id] * self.tree_.n_node_samples[node_id]
            return R, 1

        R_l, L_l = self._compute_subtree_stats(left)
        R_r, L_r = self._compute_subtree_stats(right)
        return R_l + R_r, L_l + L_r

    def _prune_tree(self):
        """
        Apply cost-complexity pruning with the given alpha.
        Greedy weakest-link strategy:
            repeatedly prune the node t with smallest g(t)
            as long as g(t) <= alpha.
        """
        alpha = self.alpha
        if alpha <= 0.0:
            return

        while True:
            n_nodes = self.tree_.node_count
            R_subtree = np.zeros(n_nodes, dtype=np.float64)
            n_leaves = np.zeros(n_nodes, dtype=np.int64)

            def dfs(node_id):
                left = self.tree_.children_left[node_id]
                right = self.tree_.children_right[node_id]
                if left == -1 and right == -1:
                    R = self.tree_.impurity[node_id] * self.tree_.n_node_samples[node_id]
                    R_subtree[node_id] = R
                    n_leaves[node_id] = 1
                    return R, 1
                R_l, L_l = dfs(left)
                R_r, L_r = dfs(right)
                R_subtree[node_id] = R_l + R_r
                n_leaves[node_id] = L_l + L_r
                return R_l + R_r, L_l + L_r

            dfs(0)

            g = np.full(n_nodes, np.inf, dtype=np.float64)
            for node_id in range(n_nodes):
                left = self.tree_.children_left[node_id]
                right = self.tree_.children_right[node_id]
                if left == -1 and right == -1:
                    continue  # leaf
                if n_leaves[node_id] <= 1:
                    continue

                R_leaf = self.tree_.impurity[node_id] * self.tree_.n_node_samples[node_id]
                R_T = R_subtree[node_id]
                denom = n_leaves[node_id] - 1
                if denom <= 0:
                    continue

                g[node_id] = (R_leaf - R_T) / denom

            min_g = g.min()
            if (not np.isfinite(min_g)) or (min_g > alpha):
                break

            node_to_prune = int(np.argmin(g))
            self.tree_.children_left[node_to_prune] = -1
            self.tree_.children_right[node_to_prune] = -1
    # Prediction
    def _predict_one_proba(self, x):
        """
        Traverse the tree for a single sample x and return class probabilities.
        """
        node = 0
        while True:
            feature = self.tree_.feature[node]
            if feature == -1:
                counts = self.tree_.value[node]
                total = counts.sum()
                if total == 0:
                    return np.ones(self.n_classes_) / self.n_classes_
                return counts / total

            thr = self.tree_.threshold[node]
            if x[feature] <= thr:
                node = self.tree_.children_left[node]
            else:
                node = self.tree_.children_right[node]

    def predict_proba(self, X):
        X = np.asarray(X, dtype=np.float64)
        if X.ndim == 1:
            X = X.reshape(1, -1)

        n_samples = X.shape[0]
        proba = np.zeros((n_samples, self.n_classes_), dtype=np.float64)
        for i in range(n_samples):
            proba[i] = self._predict_one_proba(X[i])
        return proba

    def predict(self, X):
        proba = self.predict_proba(X)
        class_indices = np.argmax(proba, axis=1)
        return self.classes_[class_indices]

    def loss(self, X, y):
        """
        Compute misclassification loss on (X, y).
        """
        y_pred = self.predict(X)
        return np.mean(y_pred != y)

    def accuracy(self, X, y):
        """
        Compute accuracy on (X, y).
        """
        y_pred = self.predict(X)
        return np.mean(y_pred == y)
```

Part 3: Check Model

In this section, we design unit tests for our `DecisionTreeCART` implementation and compare it against `sklearn.tree.DecisionTreeClassifier` on a public dataset (the breast cancer dataset). The goals are:

- verify that each method of our class works correctly in isolation,
- check that edge cases are handled properly,
- and demonstrate that our implementation can successfully reproduce sklearn's CART results.

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
import random

# 1. our CART wrapper
def make_our_cart(X_train,X_test,y_train,y_test,*, criterion="gini",max_depth=None,
                  min_sample_split=2,alpha=0.0,random_state=None,verbose=True):
    """
    Train our numpy CART on given train/test split and print acc/loss.
    loss = 1 - accuracy (0-1 loss)
    """
    clf = CARTClassifier(
        criterion=criterion,
        max_depth=max_depth,
        min_sample_split=min_sample_split,
        alpha=alpha,
        random_state=random_state,
    )
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    acc = clf.accuracy(X_test, y_test)
    loss = clf.loss(X_test, y_test)
    if verbose:
        print(f"[OUR CART] acc={acc:.4f}, loss={loss:.4f}")
    return clf, acc, loss, y_pred


# 2. sklearn CART wrapper
def make_sk_cart(X_train,X_test,y_train,y_test,*,criterion="gini",max_depth=None,
                 min_sample_split=2,random_state=None,verbose=True):
    """
    Train sklearn's DecisionTreeClassifier and print acc/loss.
    """
    sk_clf = DecisionTreeClassifier(
        criterion=criterion,
        max_depth=max_depth,
        min_samples_split=min_sample_split,
        random_state=random_state,
    )
    sk_clf.fit(X_train, y_train)
    y_pred = sk_clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    loss = 1.0 - acc
    if verbose:
        print(f"[SK CART ] acc={acc:.4f}, loss={loss:.4f}")
    return sk_clf, acc, loss, y_pred

# 3. test_on_dataset(): loop over seeds, compare our vs sklearn
def test_on_dataset(X,y,seed_list,*,criterion="gini",max_depth=None,
                    min_sample_split=2,alpha=0.0,test_size=0.3):
    """
    For each random seed:
      - create the same train/test split
      - train our CART and sklearn CART
      - compare accuracy and loss
      - print mismatched predictions (our vs sklearn) on X_test
    """
    print("==============================================")
    print("Testing on dataset with seeds:", seed_list)
    print("criterion =", criterion,
          "max_depth =", max_depth,
          "min_sample_split =", min_sample_split,
          "alpha =", alpha)
    print("==============================================\n")

    for seed in seed_list:
        print(f"--- Seed = {seed} ---")
        X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=test_size,random_state=seed,stratify=y)

        # our CART
        our_clf, our_acc, our_loss, y_pred_our = make_our_cart(X_train,X_test,y_train,y_test,criterion=criterion,
                                                               max_depth=max_depth,min_sample_split=min_sample_split,
                                                               alpha=alpha,random_state=seed,verbose=True)

        # sklearn CART
        sk_clf, sk_acc, sk_loss, y_pred_sk = make_sk_cart(X_train,X_test,y_train,y_test,criterion=criterion,
                                                          max_depth=max_depth,min_sample_split=min_sample_split,
                                                          random_state=seed,verbose=True)

        # compare
        acc_diff = our_acc - sk_acc
        loss_diff = our_loss - sk_loss
        print(f"Diff: acc (our - sk) = {acc_diff:+.4f}, "
              f"loss (our - sk) = {loss_diff:+.4f}")

        mismatch_mask = (y_pred_our != y_pred_sk)
        mismatch_idx = np.where(mismatch_mask)[0]

        if mismatch_idx.size == 0:
            print("  [MATCH] our predictions == sklearn predictions on all test samples.\n")
        else:
            print(f"  [MISMATCH] {mismatch_idx.size} samples have different predictions:")
            for i in mismatch_idx:
                print(f"    test_idx={i}: "
                      f"y_true={y_test[i]}, "
                      f"y_our={y_pred_our[i]}, "
                      f"y_sk={y_pred_sk[i]}")
            print()

    print("Done.\n")
```

Test 1–3: Basic functionality

- **Test 1 – fit():** check that training runs without error.
- **Test 2 – predict():** check output shape, label range, and report train/test accuracy.
- **Test 3 – loss():** check that `loss` returns a finite scalar (misclassification error in [0, 1]).

```
In [ ]: from sklearn.datasets import make_classification
        from sklearn.model_selection import train_test_split
        X, y = make_classification(random_state=42)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
```

```
In [17]: # Test 1
         clf = CARTClassifier()
         clf.fit(X_train, y_train)
         print("Test 1 passed: train() runs without error.")
```

```
Test 1 passed: train() runs without error.
```

```
In [ ]: # Test 2: predict() should produce outputs with correct shape and valid class values
        clf = CARTClassifier()
        clf.fit(X_train, y_train)
        y_pred_train = clf.predict(X_train)

        # Check output shape
        assert y_pred_train.shape == y_train.shape, f"Prediction shape mismatch: y_pred shape={y_pred_train.shape}, y_train shape={y_train.shape}"

        # Check value range (breast_cancer is a binary classification dataset)
        unique_vals = np.unique(y_pred_train)
        assert set(unique_vals).issubset({0, 1}),  f"Predicted values must be 0/1. Found values: {unique_vals}"

        train_acc = clf.accuracy(X_train, y_train)
        print(f"Test 2 passed: predict() shape & value checks passed. Train accuracy = {train_acc:.3f}")

        test_acc = clf.accuracy(X_test, y_test)
        print(f"Test accuracy = {test_acc:.3f}")
```

```
Test 2 passed: predict() shape & value checks passed. Train accuracy = 1.000
Test accuracy = 0.900
```

```
In [25]: # Test 3: loss() should return a finite scalar value

         clf = CARTClassifier()
         clf.fit(X_train, y_train)

         train_loss = clf.loss(X_train, y_train)

         assert np.isscalar(train_loss), "loss() should return a scalar value."
         assert np.isfinite(train_loss), "loss() should not return NaN or infinity."

         print(f"Test 3 passed: loss() returns a valid finite scalar. Train loss = {train_loss:.6f}")
```

```
Test 3 passed: loss() returns a valid finite scalar. Train loss = 0.000000
```

Our loss function is defined as the misclassification error rate, therefore it should be a scalar between 0 and 1.

Test 4: Edge cases and method-level tests

We use small toy datasets to verify that our CART implementation behaves correctly under extreme scenarios and that core methods work as intended.

Edge cases

- **Test 4.1 – All labels identical (only one class)**
- **Test 4.2 – Single feature only**
- **Test 4.3 – All-zero features**

Method-level tests

- **Test 4.4 – `predict_proba()`** : correct shape, valid probability distribution, and consistency with `predict`
- **Test 4.5 – `accuracy()`** : matches manual computation on a tiny dataset

In [ ]:
```
# A small toy dataset for edge case testing
X_toy = np.array([
    [0.0, 0.0],
    [0.0, 1.0],
    [1.0, 0.0],
    [1.0, 1.0],
])
y_toy = np.array([0, 0, 1, 1])
print("X_toy:\n", X_toy)
print("y_toy:", y_toy)
```

```
X_toy:
 [[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]
y_toy: [0 0 1 1]
```

In [27]:
```
# Test 4.1: all labels are zero (only one class present)
clf_zero = CARTClassifier()

y_all_zero = np.zeros_like(y_toy)
clf_zero.fit(X_toy, y_all_zero)

y_pred_zero = clf_zero.predict(X_toy)
loss_zero = clf_zero.loss(X_toy, y_all_zero)

assert y_pred_zero.shape == y_all_zero.shape
assert np.isfinite(loss_zero)

print("Test 4.1 passed: all-zero labels edge case handled correctly.")
print("Predicted labels:", y_pred_zero)
print("Loss on all-zero labels:", loss_zero)
```

```
Test 4.1 passed: all-zero labels edge case handled correctly.
Predicted labels: [0 0 0 0]
Loss on all-zero labels: 0.0
```

In [28]:
```
# Test 4.2: dataset contains only one feature

model_single = CARTClassifier()

X_single = X_toy[:, :1]  # Use only the first feature
model_single.fit(X_single, y_toy)

y_pred_single = model_single.predict(X_single)
assert y_pred_single.shape == y_toy.shape

print("Test 4.2 passed: single-feature edge case handled correctly.")

assert np.array_equal(y_pred_single, y_toy)
print("Predicted labels:", y_pred_single)
```

```
Test 4.2 passed: single-feature edge case handled correctly.
Predicted labels: [0 0 1 1]
```

In [29]:
```
# Test 4.3: all feature values are zero

model_feat_zero = CARTClassifier()

X_zeros = np.zeros_like(X_toy)
model_feat_zero.fit(X_zeros, y_toy)

y_pred_zeros = model_feat_zero.predict(X_zeros)
loss_zeros = model_feat_zero.loss(X_zeros, y_toy)

assert y_pred_zeros.shape == y_toy.shape
assert np.isfinite(loss_zeros)

print("Test 4.3 passed: all-zero features edge case handled correctly.")
print("Predicted labels:", y_pred_zeros)
print("Loss on all-zero features:", loss_zeros)
```

```
Test 4.3 passed: all-zero features edge case handled correctly.
Predicted labels: [0 0 0 0]
Loss on all-zero features: 0.5
```

In [34]:
```
# Test 4.4: predict_proba() shape and probabilities

model = CARTClassifier(max_depth=5, min_sample_split=2, criterion='gini')
model.fit(X_train, y_train)
proba = model.predict_proba(X_test)
n_classes = len(np.unique(y_train))
assert proba.shape == (X_test.shape[0], n_classes),f"predict_proba shape {proba.shape} does not match (n_samples, n_classes)."

row_sums = proba.sum(axis=1)
assert np.allclose(row_sums, 1.0, atol=1e-12), "Each row of predict_proba should sum to 1."

y_pred_from_proba = np.argmax(proba, axis=1)
y_pred = model.predict(X_test)
assert np.array_equal(y_pred_from_proba, y_pred), "argmax over predict_proba should match predict()."
print("Test 4b passed: predict_proba has correct shape, rows sum to 1, and argmax matches predict().")
```

```
Test 4b passed: predict_proba has correct shape, rows sum to 1, and argmax matches predict().
```

In [37]:
```
# Test 4.5: accuracy() matches manual computation
X_toy_small = np.array([[0], [1], [2], [3]])
y_toy_small = np.array([0, 0, 1, 1])

model = CARTClassifier(max_depth=2, min_sample_split=2, criterion='gini')
model.fit(X_toy_small, y_toy_small)

y_pred_toy = model.predict(X_toy_small)
manual_acc = np.mean(y_pred_toy == y_toy_small)
model_acc = model.accuracy(X_toy_small, y_toy_small)

assert np.isclose(manual_acc, model_acc), f"Manual accuracy {manual_acc} does not match model.accuracy {model_acc}."

print("Test 4c passed: model.accuracy matches manual accuracy on the toy dataset.")
```

```
Test 4c passed: model.accuracy matches manual accuracy on the toy dataset.
```

Test 5: Comparison with sklearn on a public dataset

We now compare our `CARTClassifier` implementation to `sklearn.tree.DecisionTreeClassifier` on the iris dataset.

We perform two comparisons:

- **Test 5.1 – Gini impurity**
- **Test 5.2 – Entropy impurity**
- **Test 5.3 - Gini Impurity with Pruning**
- **Test 5.4 - Entropy impurity with Pruning**

Test 5.1 Gini Impurity

- Test on multiple seeds
- `max_depth=5`
- `criterion='gini'`
- `alpha=0` : No pruning

We reached exact same accuracy as Sklearn results on these seeds, and on dataset iris and breast_cancer.

In [58]:
```
np.random.seed(0)
random.seed(0)
X_iris, y_iris = load_iris(return_X_y=True)
seeds = [0, 1, 3, 5, 10, 99]
test_on_dataset(X_iris,y_iris,seed_list=seeds,criterion="gini",max_depth=5,min_sample_split=2,alpha=0.0,test_size=0.3,)
```

```
==============================================
Testing on dataset with seeds: [0, 1, 3, 5, 10, 99]
criterion = gini max_depth = 5 min_sample_split = 2 alpha = 0.0
==============================================

--- Seed = 0 ---
[OUR CART] acc=0.9778, loss=0.0222
[SK CART ] acc=0.9778, loss=0.0222
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 1 ---
[OUR CART] acc=0.9778, loss=0.0222
[SK CART ] acc=0.9778, loss=0.0222
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 3 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 5 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 10 ---
[OUR CART] acc=1.0000, loss=0.0000
[SK CART ] acc=1.0000, loss=0.0000
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 99 ---
[OUR CART] acc=0.9778, loss=0.0222
[SK CART ] acc=0.9778, loss=0.0222
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

Done.
```

In [61]:
```
from sklearn.datasets import load_breast_cancer
np.random.seed(0)
random.seed(0)
X_breast_cancer, y_breast_cancer = load_breast_cancer(return_X_y=True)
seeds = [1,10, 55, 66, 99]
test_on_dataset(X_breast_cancer,y_breast_cancer,seed_list=seeds,criterion="gini",max_depth=5,min_sample_split=2,alpha=0.0,test_size=0.3,)
```

```
==============================================
Testing on dataset with seeds: [1, 10, 55, 66, 99]
criterion = gini max_depth = 5 min_sample_split = 2 alpha = 0.0
==============================================

--- Seed = 1 ---
[OUR CART] acc=0.9415, loss=0.0585
[SK CART ] acc=0.9415, loss=0.0585
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MISMATCH] 2 samples have different predictions:
    test_idx=126: y_true=1, y_our=1, y_sk=0
    test_idx=165: y_true=0, y_our=1, y_sk=0

--- Seed = 10 ---
[OUR CART] acc=0.9532, loss=0.0468
[SK CART ] acc=0.9532, loss=0.0468
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 55 ---
[OUR CART] acc=0.9298, loss=0.0702
[SK CART ] acc=0.9298, loss=0.0702
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 66 ---
[OUR CART] acc=0.8889, loss=0.1111
[SK CART ] acc=0.8889, loss=0.1111
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 99 ---
[OUR CART] acc=0.9240, loss=0.0760
[SK CART ] acc=0.9240, loss=0.0760
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MISMATCH] 4 samples have different predictions:
    test_idx=2: y_true=1, y_our=0, y_sk=1
    test_idx=19: y_true=0, y_our=0, y_sk=1
    test_idx=21: y_true=1, y_our=0, y_sk=1
    test_idx=118: y_true=0, y_our=0, y_sk=1

Done.
```

Test 5.2 Entropy Impurity

- Test on multiple seeds
- `max_depth=5`
- `criterion='entropy'`
- `alpha=0` : No pruning

We reached exact same accuracy as Sklearn results on these seeds.

In [62]:
```
np.random.seed(0)
random.seed(0)
X_iris, y_iris = load_iris(return_X_y=True)
seeds = [0, 1, 5, 9, 30]
test_on_dataset(X_iris,y_iris,seed_list=seeds,criterion="entropy",max_depth=None,min_sample_split=2,alpha=0.3,test_size=0.2)
```

```
=============================================
Testing on dataset with seeds: [0, 1, 5, 9, 30]
criterion = entropy max_depth = None min_sample_split = 2 alpha = 0.3
=============================================

--- Seed = 0 ---
[OUR CART] acc=0.9667, loss=0.0333
[SK CART ] acc=0.9667, loss=0.0333
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 1 ---
[OUR CART] acc=0.9667, loss=0.0333
[SK CART ] acc=0.9667, loss=0.0333
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 5 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 9 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 30 ---
[OUR CART] acc=0.9000, loss=0.1000
[SK CART ] acc=0.9000, loss=0.1000
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

Done.
```

In [72]:

```python
from sklearn.datasets import load_breast_cancer
np.random.seed(0)
random.seed(0)
X_breast_cancer, y_breast_cancer = load_breast_cancer(return_X_y=True)
seeds = [3,6,8,25,88]
test_on_dataset(X_breast_cancer,y_breast_cancer,seed_list=seeds,criterion="entropy",max_depth=5,min_sample_split=2,alpha=0.0,test_size=0.3,)
```

```
==============================================
Testing on dataset with seeds: [3, 6, 8, 25, 88]
criterion = entropy max_depth = 5 min_sample_split = 2 alpha = 0.0
==============================================

--- Seed = 3 ---
[OUR CART] acc=0.9240, loss=0.0760
[SK CART ] acc=0.9240, loss=0.0760
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MISMATCH] 4 samples have different predictions:
    test_idx=5: y_true=1, y_our=1, y_sk=0
    test_idx=32: y_true=0, y_our=0, y_sk=1
    test_idx=108: y_true=1, y_our=0, y_sk=1
    test_idx=140: y_true=1, y_our=0, y_sk=1

--- Seed = 6 ---
[OUR CART] acc=0.9649, loss=0.0351
[SK CART ] acc=0.9649, loss=0.0351
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 8 ---
[OUR CART] acc=0.9123, loss=0.0877
[SK CART ] acc=0.9123, loss=0.0877
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 25 ---
[OUR CART] acc=0.9357, loss=0.0643
[SK CART ] acc=0.9357, loss=0.0643
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 88 ---
[OUR CART] acc=0.9532, loss=0.0468
[SK CART ] acc=0.9532, loss=0.0468
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MISMATCH] 4 samples have different predictions:
    test_idx=63: y_true=0, y_our=1, y_sk=0
    test_idx=80: y_true=0, y_our=0, y_sk=1
    test_idx=98: y_true=1, y_our=0, y_sk=1
    test_idx=108: y_true=1, y_our=1, y_sk=0

Done.
```

Test 5.3 Gini Impurity with Pruning

- Test on multiple seeds
- `max_depth=10`
- `criterion='gini'`
- `alpha=0.3` : Pruning Applied

We reached exact same accuracy as Sklearn results on these seeds.

In [101…

```
# iris dataset
np.random.seed(0)
random.seed(0)
```

```
X_iris, y_iris = load_iris(return_X_y=True)
seeds = [2, 3, 5, 10, 13]
test_on_dataset(X_iris,y_iris,seed_list=seeds,criterion="gini",max_depth=10,min_sample_split=2,alpha=0.3,test_size=0.3,)
```

```
==============================================
Testing on dataset with seeds: [2, 3, 5, 10, 13]
criterion = gini max_depth = 10 min_sample_split = 2 alpha = 0.3
==============================================

--- Seed = 2 ---
[OUR CART] acc=0.9778, loss=0.0222
[SK CART ] acc=0.9778, loss=0.0222
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 3 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 5 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 10 ---
[OUR CART] acc=1.0000, loss=0.0000
[SK CART ] acc=1.0000, loss=0.0000
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 13 ---
[OUR CART] acc=0.9778, loss=0.0222
[SK CART ] acc=0.9778, loss=0.0222
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

Done.
```

In [102…

```
# breast cancer dataset
np.random.seed(0)
random.seed(0)
X_breast_cancer, y_breast_cancer = load_breast_cancer(return_X_y=True)
seeds = [1,6,7,10,13]
test_on_dataset(X_breast_cancer,y_breast_cancer,seed_list=seeds,criterion="gini",max_depth=10,min_sample_split=2,alpha=0.3,test_size=0.3,)
```

```
==============================================
Testing on dataset with seeds: [1, 6, 7, 10, 13]
criterion = gini max_depth = 10 min_sample_split = 2 alpha = 0.3
==============================================

--- Seed = 1 ---
[OUR CART] acc=0.9415, loss=0.0585
[SK CART ] acc=0.9415, loss=0.0585
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MISMATCH] 2 samples have different predictions:
    test_idx=126: y_true=1, y_our=1, y_sk=0
    test_idx=165: y_true=0, y_our=1, y_sk=0

--- Seed = 6 ---
[OUR CART] acc=0.9649, loss=0.0351
[SK CART ] acc=0.9649, loss=0.0351
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 7 ---
[OUR CART] acc=0.9532, loss=0.0468
[SK CART ] acc=0.9532, loss=0.0468
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MISMATCH] 2 samples have different predictions:
    test_idx=96: y_true=1, y_our=0, y_sk=1
    test_idx=131: y_true=0, y_our=0, y_sk=1

--- Seed = 10 ---
[OUR CART] acc=0.9415, loss=0.0585
[SK CART ] acc=0.9415, loss=0.0585
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MISMATCH] 2 samples have different predictions:
    test_idx=43: y_true=0, y_our=1, y_sk=0
    test_idx=157: y_true=1, y_our=1, y_sk=0

--- Seed = 13 ---
[OUR CART] acc=0.9181, loss=0.0819
[SK CART ] acc=0.9181, loss=0.0819
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

Done.
```

Test 5.4 Entropy Impurity with Pruning

- Test on multiple seeds
- `max_depth=10`
- `criterion='entropy'`
- `alpha=0.3` : Pruning Applied

We reached exact same accuracy as Sklearn results on these seeds.

In [103…

```python
# iris dataset
np.random.seed(0)
random.seed(0)
X_iris, y_iris = load_iris(return_X_y=True)
```

```
seeds = [2, 3, 5, 10, 13]
test_on_dataset(X_iris,y_iris,seed_list=seeds,criterion="entropy",max_depth=10,min_sample_split=2,alpha=0.3,test_size=0.3,)
```

```
==============================================
Testing on dataset with seeds: [2, 3, 5, 10, 13]
criterion = entropy max_depth = 10 min_sample_split = 2 alpha = 0.3
==============================================

--- Seed = 2 ---
[OUR CART] acc=0.9778, loss=0.0222
[SK CART ] acc=0.9778, loss=0.0222
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 3 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 5 ---
[OUR CART] acc=0.9333, loss=0.0667
[SK CART ] acc=0.9333, loss=0.0667
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 10 ---
[OUR CART] acc=1.0000, loss=0.0000
[SK CART ] acc=1.0000, loss=0.0000
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 13 ---
[OUR CART] acc=0.9556, loss=0.0444
[SK CART ] acc=0.9556, loss=0.0444
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

Done.
```

In [104…

```
# breast cancer dataset
np.random.seed(0)
random.seed(0)
X_breast_cancer, y_breast_cancer = load_breast_cancer(return_X_y=True)
seeds = [3,6,21,42]
test_on_dataset(X_breast_cancer,y_breast_cancer,seed_list=seeds,criterion="entropy",max_depth=10,min_sample_split=2,alpha=0.3,test_size=0.3,)
```

```
==============================================
Testing on dataset with seeds: [3, 6, 21, 42]
criterion = entropy max_depth = 10 min_sample_split = 2 alpha = 0.3
==============================================

--- Seed = 3 ---
[OUR CART] acc=0.9240, loss=0.0760
[SK CART ] acc=0.9240, loss=0.0760
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MISMATCH] 4 samples have different predictions:
    test_idx=5: y_true=1, y_our=1, y_sk=0
    test_idx=32: y_true=0, y_our=0, y_sk=1
    test_idx=108: y_true=1, y_our=0, y_sk=1
    test_idx=140: y_true=1, y_our=0, y_sk=1

--- Seed = 6 ---
[OUR CART] acc=0.9649, loss=0.0351
[SK CART ] acc=0.9649, loss=0.0351
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MATCH] our predictions == sklearn predictions on all test samples.

--- Seed = 21 ---
[OUR CART] acc=0.9064, loss=0.0936
[SK CART ] acc=0.9064, loss=0.0936
Diff: acc (our - sk) = +0.0000, loss (our - sk) = +0.0000
  [MISMATCH] 2 samples have different predictions:
    test_idx=132: y_true=1, y_our=0, y_sk=1
    test_idx=150: y_true=1, y_our=1, y_sk=0

--- Seed = 42 ---
[OUR CART] acc=0.9474, loss=0.0526
[SK CART ] acc=0.9474, loss=0.0526
Diff: acc (our - sk) = +0.0000, loss (our - sk) = -0.0000
  [MISMATCH] 4 samples have different predictions:
    test_idx=13: y_true=0, y_our=1, y_sk=0
    test_idx=17: y_true=1, y_our=0, y_sk=1
    test_idx=30: y_true=1, y_our=1, y_sk=0
    test_idx=104: y_true=1, y_our=1, y_sk=0

Done.
```

Test 6: Node impurity calculation

Finally, we directly unit-test our impurity functions `node_score_gini` and `node_score_entropy` against sklearn's impurity values on several label distributions:

- pure node (all labels identical),
- balanced 50/50 node,
- skewed binary labels,
- multi-class labels.

We construct a root-only sklearn tree and compare the impurity stored at the root to our implementation (up to a log-base factor for entropy).

In [110…

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier

def test_node_impurity_against_sklearn():
    """
    Unit-test node_score_gini_from_counts and node_score_entropy_from_counts
    against sklearn's root impurity on several label distributions.
    """
    def sklearn_root_impurity(y, criterion):
        """
        Fit a root-only sklearn tree and return its root impurity.
        """
        X_dummy = np.zeros((len(y), 1))  # dummy feature
        clf = DecisionTreeClassifier(
            criterion=criterion,
            max_depth=1,          # root only
            random_state=0
        )
        clf.fit(X_dummy, y)
        return float(clf.tree_.impurity[0])

    def our_impurity(y, criterion):
        """
        Compute impurity using our counts-based implementation.
        """
        classes = np.unique(y)
        mapping = {c: i for i, c in enumerate(classes)}
        y_enc = np.array([mapping[c] for c in y], dtype=np.int64)
        counts = np.bincount(y_enc, minlength=len(classes))

        if criterion == "gini":
            return node_score_gini_from_counts(counts)
        elif criterion == "entropy":
            return node_score_entropy_from_counts(counts)

    # Test cases: (description, labels)
    test_cases = [
        ("pure node",        np.array([0, 0, 0, 0, 0])),
        ("balanced 50/50",   np.array([0, 0, 1, 1])),
        ("skewed binary",    np.array([0, 0, 0, 1])),
        ("multi-class",      np.array([0, 1, 2, 0, 1, 2])),
    ]

    for name, y in test_cases:
        print(f"--- {name} ---")
        # Gini
        sk_gini = sklearn_root_impurity(y, criterion="gini")
        our_gini = our_impurity(y, criterion="gini")
        print(f"  Gini:    sklearn={sk_gini:.6f}, ours={our_gini:.6f}")
        assert np.isclose(our_gini, sk_gini, atol=1e-12), f"Gini mismatch for case '{name}'"

        # Entropy
        sk_entropy = sklearn_root_impurity(y, criterion="entropy")
        our_entropy = our_impurity(y, criterion="entropy")
        print(f"  Entropy: sklearn={sk_entropy:.6f}, ours={our_entropy:.6f}")
        assert np.isclose(our_entropy, sk_entropy, atol=1e-12), f"Entropy mismatch for case '{name}'"

    print("\nAll impurity tests PASSED")

test_node_impurity_against_sklearn()
```

```
--- pure node ---
  Gini:    sklearn=0.000000, ours=0.000000
  Entropy: sklearn=0.000000, ours=0.000000
--- balanced 50/50 ---
  Gini:    sklearn=0.500000, ours=0.500000
  Entropy: sklearn=1.000000, ours=1.000000
--- skewed binary ---
  Gini:    sklearn=0.375000, ours=0.375000
  Entropy: sklearn=0.811278, ours=0.811278
--- multi-class ---
  Gini:    sklearn=0.666667, ours=0.666667
  Entropy: sklearn=1.584963, ours=1.584963

All impurity tests PASSED
```

References

1. scikit-learn developers (2024) *Decision Trees: Mathematical Formulation*. Available at: https://scikit-learn.org/stable/modules/tree.html#tree-mathematical-formulation.

2. Breiman, L., Friedman, J., Olshen, R. and Stone, C., 1984. Classification and Regression Trees. Belmont, CA: Wadsworth.

3. Quinlan, J. R. (1993). C4.5: Programs for Machine Learning. Morgan Kaufmann.

4. Irizarry, R., 2023. Data Science: Decision Trees (Section 11). Harvard T.H. Chan School of Public Health. Available at: https://rafalab.dfci.harvard.edu/pages/649/section-11.pdf