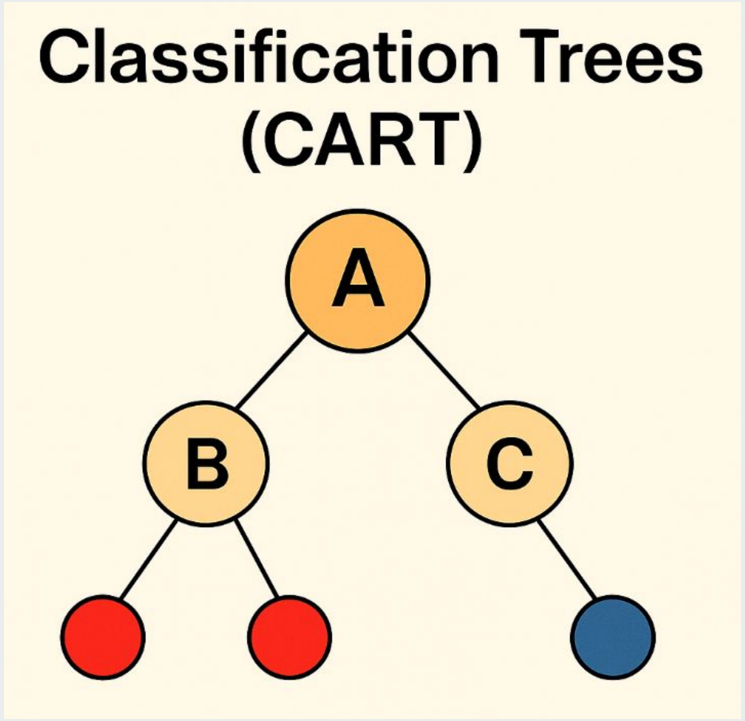
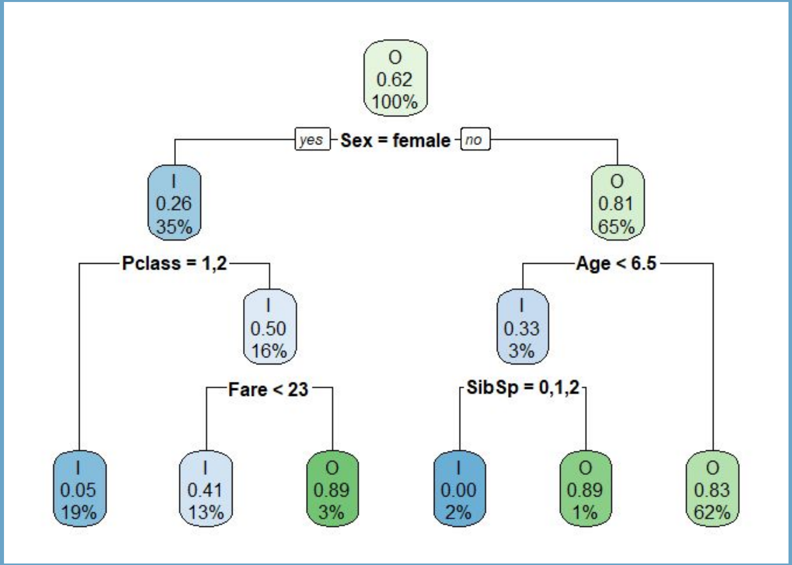
CART for Classification

Mingrui Chen
Muxin Fu
Jingmin Xu
Yixiao Zhang



Meet CART 🌲

- CART (Classification and Regression Trees) is a foundational supervised learning algorithm used for classification and regression tasks.
- It works by recursively partitioning the feature space into regions that are increasingly "pure" with respect to the target variable.
- The algorithm selects splits based on impurity measures (e.g., Gini index, entropy) to build an interpretable tree structure.
- CART remains widely used today due to its interpretability, flexibility, and role in ensemble models (Random Forests, Gradient Boosting).



Reference:https://machinelearning-basics.com/cart-algorithm-and-everything-you-need-to-know-about-decision-trees-1-2/

Advantages & Disadvantages

Advantages:

- Allow split on **continuous** features (!)
  - Threshold-based Split
- Highly interpretable
- Captures nonlinear relationships
- Minimal preprocessing required

Disadvantages:

- Prone to overfitting
- Unstable
- Limited predictive power compared with ensembles

Math behind the ML algorithm

Part 1: Representation

Part 1: Representation

a. Domain Set $\mathcal{X} = \mathbb{R}^n, \quad x_i = (x_{i1}, x_{i2}, \ldots, x_{in}) \in \mathcal{X}.$

b. Label Set $\mathcal{Y} = \{0, 1, \ldots, K-1\}.$

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

d. Learner's Output $\mathcal{H} = \{\, h : \mathcal{X} \to \mathcal{Y} \mid h \text{ is a binary decision tree with depth } \leq T_{\max} \,\}.$

Model as a Hierarchical Partition

- Each split defined by a feature *j* and threshold *t:*

$$\begin{array}{lll} x_j \leq t & \Rightarrow & \text{left child} \\ x_j > t & \Rightarrow & \text{right child} \end{array}$$

Math behind the ML algorithm

Part 2: Loss

Part 2.1: Loss - Impurity Measure

Gini:

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

Entropy:

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

a. Node *i* containing a subset of samples:

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

b. The number of samples in node *i* that belongs to class k:

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

c. Class proportion of class k in node i:

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

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

Part 2.2: Loss - Split Loss

CART chooses the split that minimizes impurity of child nodes.

- The general **objective** therefore is:

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

- At node i, the dataset *S_i* is partitioned into a left subset and right subset

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

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

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

- It calculated a weighted child impurity
- C(*) is the impurity measure function

Math behind the ML algorithm

Part 3: Optimization

Part 3.1: Greedy Search

- CART optimize to select the parameters that minimises the impurity at each split:

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

- **Greedy Optimization**
  - For each node,
    - Evaluate every feature *j*
    - Evaluate every candidate threshold *t*
    - Compute impurity of left/right partitions
    - Choose the best pair
  - For example, at *node i*, we have impurity measure:

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

Part 3.2: Threshold & Stopping

- **Threshold candidate generation:**
  - For sorted feature column, take **midpoints** as threshold candidates
  - For feature column $v_1 < v_2 < \cdots < v_m$, we have:,

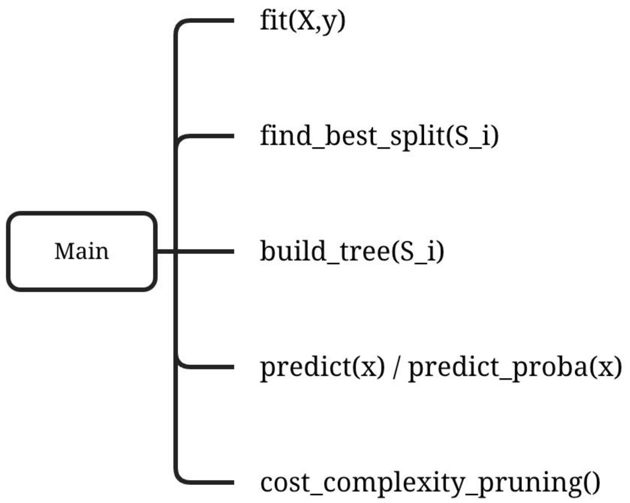
$$\mathcal{T}_j = \left\{ t_k = \frac{v_k + v_{k+1}}{2} \;\middle|\; k = 1, \ldots, m-1 \right\}$$

  - Example:
    - For feature j, it has values: $\{1, 1, 2, 2, 4\}$
    - Unique sorted : $\{1,2,4\}$
    - Candidate thresholds: $t1 = 1.5$, $t2 = 3.0$
- **Stopping Criterion:**

$$\mathrm{Stop}(S_i) = \Big(\underbrace{c(S_i) = 0}_{\text{pure node}}\Big) \vee \Big(\underbrace{|S_i| < \text{min\_samples\_split}}_{\text{too few samples}}\Big) \vee \Big(\underbrace{\text{depth}(S_i) \geq \text{max\_depth}}_{\text{depth limit}}\Big) \vee \Big(\underbrace{\Theta_i = \varnothing}_{\text{no valid split}}\Big)$$

Numerical Techniques

Part 1: Numerical Structure



Part 2.1: find the best split

```
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)
```

—> Evaluate each feature as a candidate

—> Generate threshold midpoints

—> Evaluate each threshold, and update the best split

- find the best feature-threshold pair (j^*, t^*) minimizing node loss.
- Return best_feature, best_threshold for each node

Part 2.2: build_tree recursively

```
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
           )
```

→ Stopping Conditions

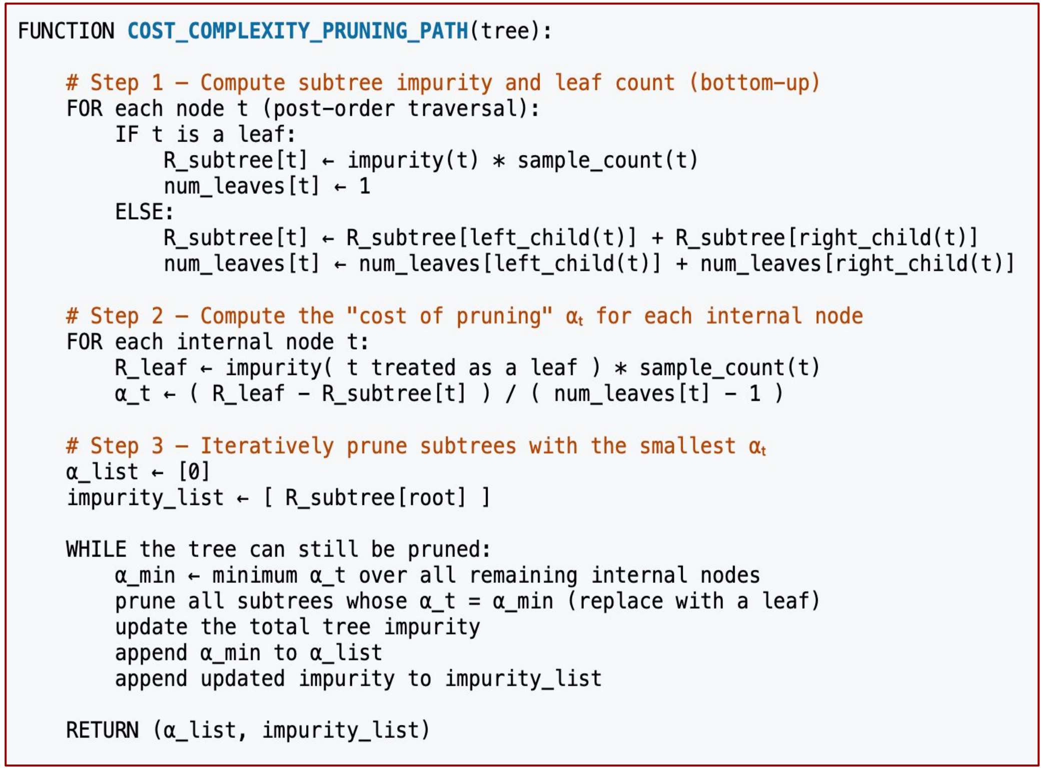
→ Find best split parameters (feature & threshold)

→ Split dataset, and grow recursively

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

- Sklearn has more details constraints, we only implement the main controls and ignore some of it (like splitter, use *splitter* = *best*, assume *min_impurity_decrease=0.0* etc)

Part 2.2: cost_complexity pruning

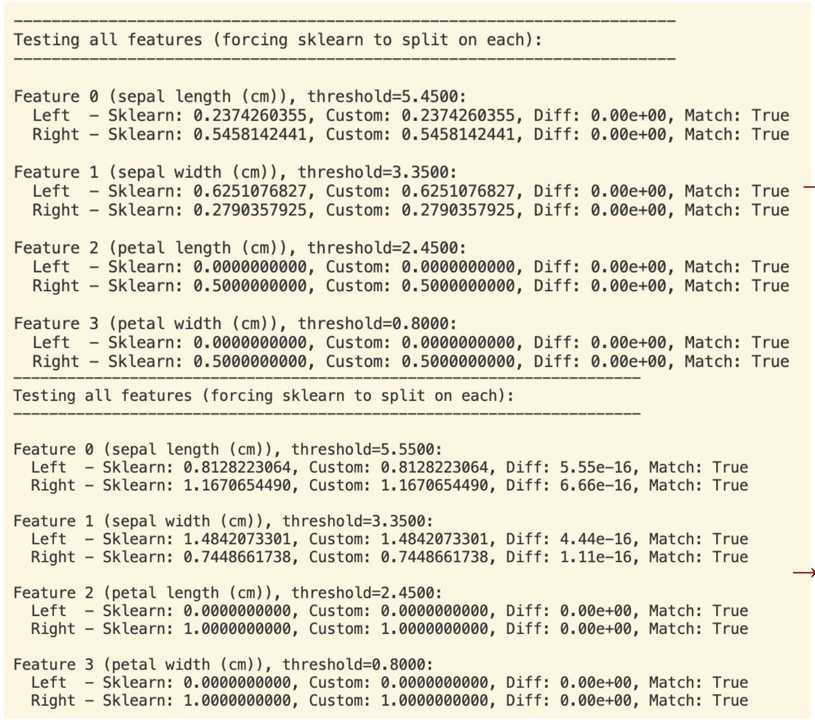


```
FUNCTION COST_COMPLEXITY_PRUNING_PATH(tree):

    # 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" α_t 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 α_t
    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)
```

- Larger alpha -> Stronger penalty -> More pruning
- Default alpha = 0, full tree
- The main idea is what's the cost of each internal node, if we cut it
- Cost can be understood as the increase of impurity / number of removed leaves

Sklearn Results

1. Impurity Measure



```
----------------------------------------------------------------------
Testing all features (forcing sklearn to split on each):
----------------------------------------------------------------------

Feature 0 (sepal length (cm)), threshold=5.4500:
  Left  - Sklearn: 0.2374260355, Custom: 0.2374260355, Diff: 0.00e+00, Match: True
  Right - Sklearn: 0.5458142441, Custom: 0.5458142441, Diff: 0.00e+00, Match: True

Feature 1 (sepal width (cm)), threshold=3.3500:
  Left  - Sklearn: 0.6251076827, Custom: 0.6251076827, Diff: 0.00e+00, Match: True
  Right - Sklearn: 0.2790357925, Custom: 0.2790357925, Diff: 0.00e+00, Match: True

Feature 2 (petal length (cm)), threshold=2.4500:
  Left  - Sklearn: 0.0000000000, Custom: 0.0000000000, Diff: 0.00e+00, Match: True
  Right - Sklearn: 0.5000000000, Custom: 0.5000000000, Diff: 0.00e+00, Match: True

Feature 3 (petal width (cm)), threshold=0.8000:
  Left  - Sklearn: 0.0000000000, Custom: 0.0000000000, Diff: 0.00e+00, Match: True
  Right - Sklearn: 0.5000000000, Custom: 0.5000000000, Diff: 0.00e+00, Match: True
----------------------------------------------------------------------
Testing all features (forcing sklearn to split on each):
----------------------------------------------------------------------

Feature 0 (sepal length (cm)), threshold=5.5500:
  Left  - Sklearn: 0.8128223064, Custom: 0.8128223064, Diff: 5.55e-16, Match: True
  Right - Sklearn: 1.1670654490, Custom: 1.1670654490, Diff: 6.66e-16, Match: True

Feature 1 (sepal width (cm)), threshold=3.3500:
  Left  - Sklearn: 1.4842073301, Custom: 1.4842073301, Diff: 4.44e-16, Match: True
  Right - Sklearn: 0.7448661738, Custom: 0.7448661738, Diff: 1.11e-16, Match: True

Feature 2 (petal length (cm)), threshold=2.4500:
  Left  - Sklearn: 0.0000000000, Custom: 0.0000000000, Diff: 0.00e+00, Match: True
  Right - Sklearn: 1.0000000000, Custom: 1.0000000000, Diff: 0.00e+00, Match: True

Feature 3 (petal width (cm)), threshold=0.8000:
  Left  - Sklearn: 0.0000000000, Custom: 0.0000000000, Diff: 0.00e+00, Match: True
  Right - Sklearn: 1.0000000000, Custom: 1.0000000000, Diff: 0.00e+00, Match: True
```

→ Gini

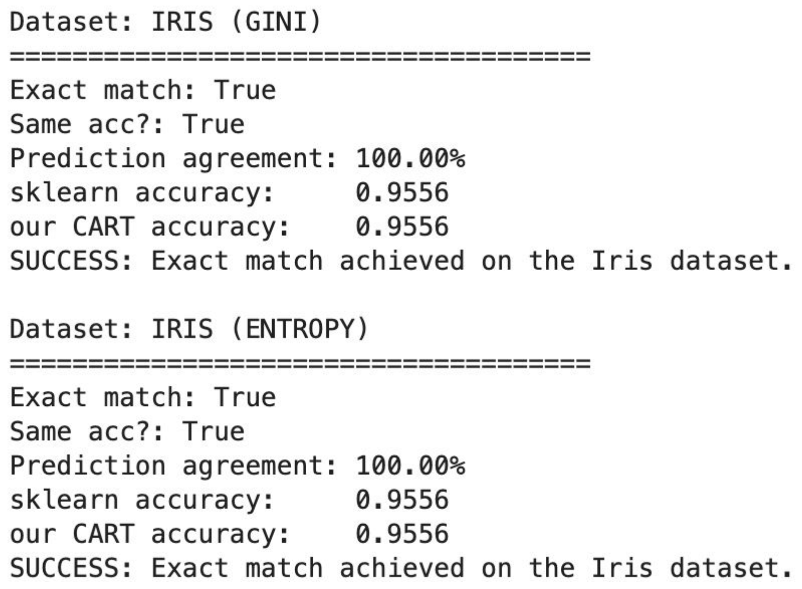
→ Entropy

- We have the **same impurity measure** result at each feature compared with sklearn
- This test was created to make sure floating precision (to check **tie-breaking** problem later on)

2. Performance: seed = 0

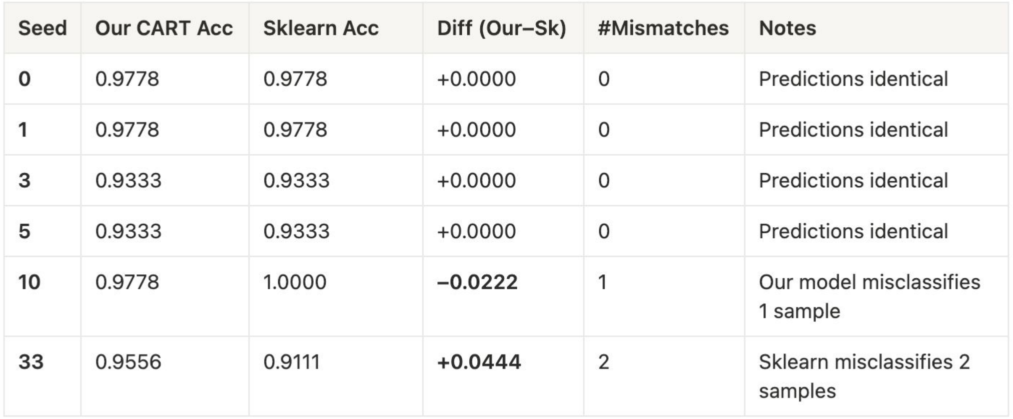
Experimental Setup

- Dataset: Iris
- Train/Test Split: 70% / 30%, stratified, random_state = 0
- Models Compared:
  - sklearn.tree.DecisionTreeCl assifier
  - Our DecisionTreeCART implementation



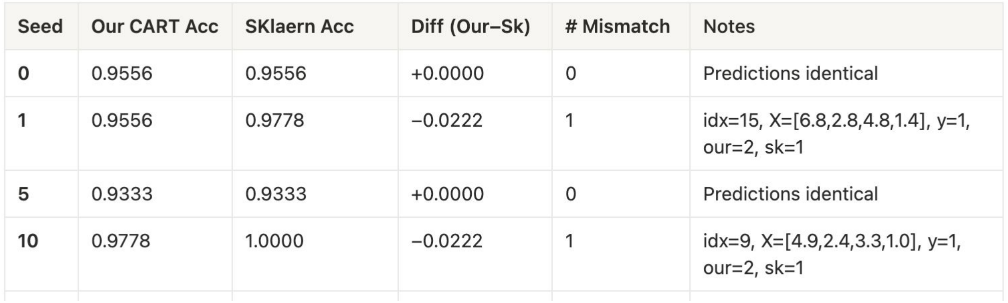
```
Dataset: IRIS (GINI)
=====================================
Exact match: True
Same acc?: True
Prediction agreement: 100.00%
sklearn accuracy:     0.9556
our CART accuracy:    0.9556
SUCCESS: Exact match achieved on the Iris dataset.

Dataset: IRIS (ENTROPY)
=====================================
Exact match: True
Same acc?: True
Prediction agreement: 100.00%
sklearn accuracy:     0.9556
our CART accuracy:    0.9556
SUCCESS: Exact match achieved on the Iris dataset.
```

2. Performance: More



Seed	Our CART Acc	Sklearn Acc	Diff (Our–Sk)	#Mismatches	Notes
0	0.9778	0.9778	+0.0000	0	Predictions identical
1	0.9778	0.9778	+0.0000	0	Predictions identical
3	0.9333	0.9333	+0.0000	0	Predictions identical
5	0.9333	0.9333	+0.0000	0	Predictions identical
10	0.9778	1.0000	**−0.0222**	1	Our model misclassifies 1 sample
33	0.9556	0.9111	**+0.0444**	2	Sklearn misclassifies 2 samples

→ Gini



Seed	Our CART Acc	SKlaern Acc	Diff (Our–Sk)	# Mismatch	Notes
0	0.9556	0.9556	+0.0000	0	Predictions identical
1	0.9556	0.9778	−0.0222	1	idx=15, X=[6.8,2.8,4.8,1.4], y=1, our=2, sk=1
5	0.9333	0.9333	+0.0000	0	Predictions identical
10	0.9778	1.0000	−0.0222	1	idx=9, X=[4.9,2.4,3.3,1.0], y=1, our=2, sk=1

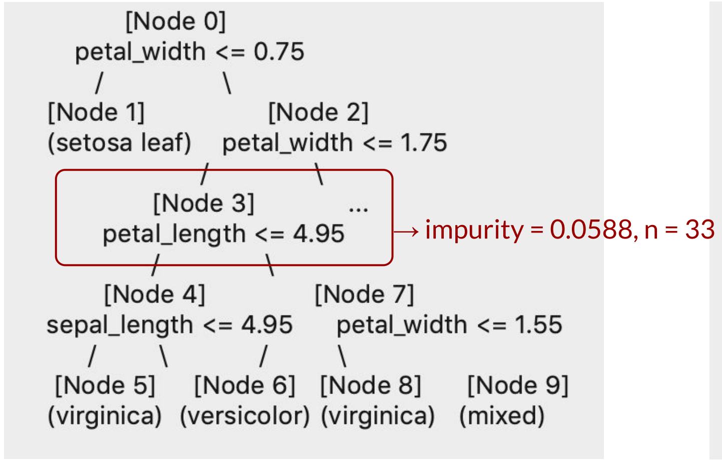
→ Entropy

Experimental Setup

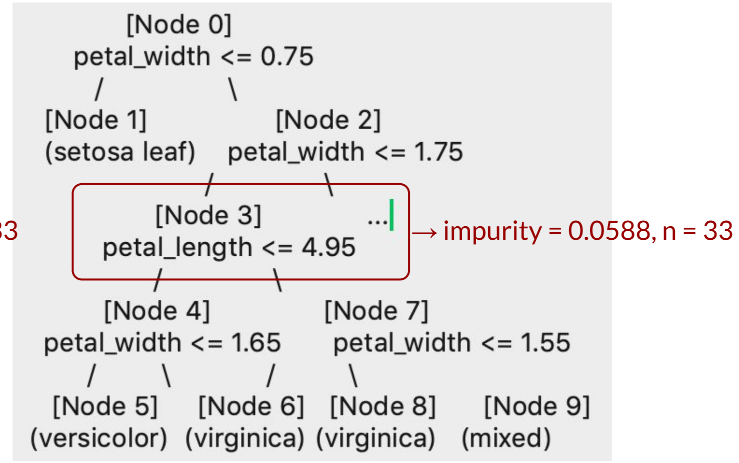
- Dataset: Iris
- Train/Test Split: 70% / 30%
- Max_depth = 5
- Min_samples_split = 2

- We reached almost the same prediction and split as what sklearn does
- Only with minor mismatch due

3. Tie-breaking Condition



Ours



Sklearn

- We future explore the case when seed=10, why we have one mismatch
- We found that it happens at the tree node 3 -> Where our model select feature 2, and sklearn select feature 3 to split one
- However, this two features **have the same impurity measure** -> Tie breaking condition
- This may due to the difference in how sklearn implement randomness, and how they deal with the same gain features.

Summary Slide

What We Found Interesting

- CART’s decision logic is intuitive yet elegant:
  It evaluates all features, sorts values, and picks the split with the lowest impurity.

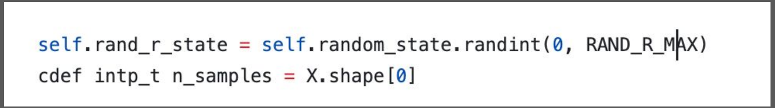
- The model is highly interpretable:
  Each node’s split can be explained visually and logically—rare among ML models.

What Was Challenging

- Debugging tree mismatches: Comparing node structures, impurity calculations, and thresholds was surprisingly demanding.

A little bit more..

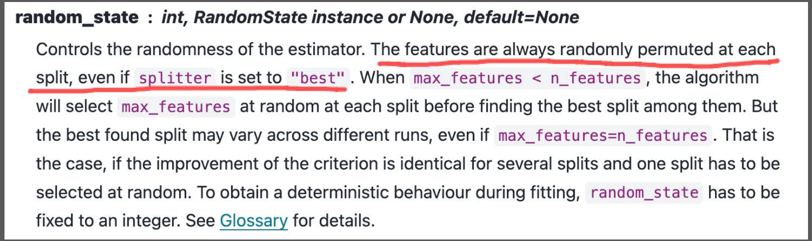
- Random State Control:
  - In Sklearn documents, it said:
  - We implement it, permute feature every split, using numpy.random.RandomState
- But the source code, written in cython, it's:



```
self.rand_r_state = self.random_state.randint(0, RAND_R_MAX)
cdef intp_t n_samples = X.shape[0]
```

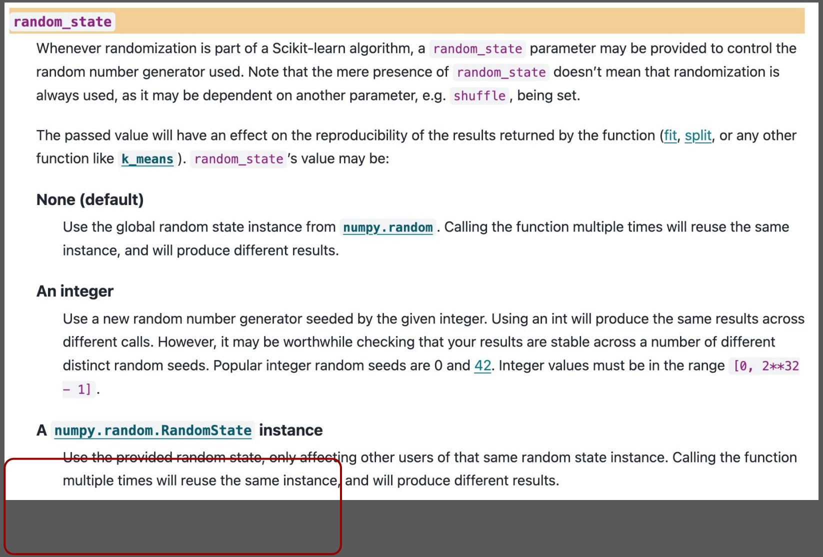
- This may lead to the different permutation order, and therefore the way how we encounter the features that have the same gain will be different
  - Since use strictly gain comparison, the first met one will be taken
- Also, float32 / float64 will have different precision.

Source Code: https://github.com/scikit-learn/scikit-learn/blob/1eb422d6c5f46a98a318f341de3e4709f9521bfe/sklearn/tree/_splitter.pyx



random_state : *int, RandomState instance or None, default=None*

Controls the randomness of the estimator. The features are always randomly permuted at each split, even if `splitter` is set to `"best"`. When `max_features < n_features`, the algorithm will select `max_features` at random at each split before finding the best split among them. But the best found split may vary across different runs, even if `max_features=n_features`. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed to an integer. See Glossary for details.



`random_state`

Whenever randomization is part of a Scikit-learn algorithm, a `random_state` parameter may be provided to control the random number generator used. Note that the mere presence of `random_state` doesn't mean that randomization is always used, as it may be dependent on another parameter, e.g. `shuffle`, being set.

The passed value will have an effect on the reproducibility of the results returned by the function (fit, split, or any other function like `k_means`). `random_state`'s value may be:

None (default)

Use the global random state instance from `numpy.random`. Calling the function multiple times will reuse the same instance, and will produce different results.

An integer

Use a new random number generator seeded by the given integer. Using an int will produce the same results across different calls. However, it may be worthwhile checking that your results are stable across a number of different distinct random seeds. Popular integer random seeds are 0 and 42. Integer values must be in the range `[0, 2**32 - 1]`.

A `numpy.random.RandomState` instance

Use the provided random state, only affecting other users of that same random state instance. Calling the function multiple times will reuse the same instance, and will produce different results.

Thank you!

Q & A