

MA289 Mathematics of AI
Decision Trees via Recursive Formulas in Python
 United States Military Academy, West Point
 Instructor: MAJ Patrick Kuiper

1 Goal and Big Picture

Goal: Understand decision trees as a *recursive partition* of feature space, and implement the core algorithm using *recursive formulas* in Python.

A decision tree represents a function of the form

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

where the regions $\{R_m\}$ are axis-aligned rectangles produced by **recursive binary splitting**. For regression, c_m is typically a mean response; for classification, c_m is a class label (and often class probabilities).

2 Regression Trees: Splits as RSS Minimization

2.1 Leaf prediction and objective

If a node contains indices $S \subset \{1, \dots, n\}$, the leaf prediction is

$$\hat{c}(S) = \frac{1}{|S|} \sum_{i \in S} y_i, \quad \text{and} \quad \text{RSS}(S) = \sum_{i \in S} (y_i - \hat{c}(S))^2.$$

A candidate split chooses a feature j and cutpoint s :

$$S_L(j, s) = \{i \in S : x_{ij} < s\}, \quad S_R(j, s) = \{i \in S : x_{ij} \geq s\}.$$

The best split at node S is

$$(j^*, s^*) = \arg \min_{j, s} \left(\text{RSS}(S_L(j, s)) + \text{RSS}(S_R(j, s)) \right).$$

2.2 Why this is recursive

Once we pick (j^*, s^*) at the current node, we repeat the same optimization *inside each child node*. This yields the **recursive definition** of the fitted tree:

$$\hat{f}_S(x) = \begin{cases} \hat{c}(S), & \text{if stopping criterion holds at node } S, \\ \hat{f}_{S_L}(x), & \text{if } x_j < s, \\ \hat{f}_{S_R}(x), & \text{if } x_j \geq s. \end{cases}$$

Typical stopping criteria: max depth reached, $|S|$ below a minimum, or no split improves the objective.

3 Classification Trees: Splits via Node Impurity

At a node S , let $\hat{p}_k(S)$ be the fraction of points in class k .

Common impurity measures:

$$\text{Classification error: } E(S) = 1 - \max_k \hat{p}_k(S),$$

$$\text{Gini: } G(S) = \sum_{k=1}^K \hat{p}_k(S) (1 - \hat{p}_k(S)),$$

$$\text{Entropy: } D(S) = - \sum_{k=1}^K \hat{p}_k(S) \log(\hat{p}_k(S)).$$

A split is chosen to maximize impurity reduction, equivalently minimize weighted impurity:

$$(j^*, s^*) = \arg \min_{j, s} \left(\frac{|S_L|}{|S|} I(S_L) + \frac{|S_R|}{|S|} I(S_R) \right),$$

where $I(\cdot)$ is Gini or entropy (more sensitive than classification error during growth).

4 Pruning: Cost-Complexity Objective

A large tree can overfit. Cost-complexity pruning selects a subtree T by minimizing

$$\text{Cost}_\alpha(T) = \sum_{m=1}^{|T|} \sum_{i: x_i \in R_m} (y_i - \hat{c}(R_m))^2 + \alpha |T|,$$

where $|T|$ is the number of terminal nodes and $\alpha \geq 0$ penalizes complexity. Pick α via cross-validation, then refit/prune using that α .

5 Python: Recursive Tree Building From Scratch (Core Formulas)

5.1 Data representation for recursion

We will represent a node as a dictionary:

- Leaf node: `{"is_leaf": True, "value": c}`
- Internal node: `{"is_leaf": False, "j": j, "s": s, "left": ..., "right": ...}`

5.2 Fast RSS computation (formula-first)

Given values $y_S = \{y_i : i \in S\}$,

$$\text{RSS}(S) = \sum_{i \in S} (y_i - \bar{y}_S)^2 = \sum_{i \in S} y_i^2 - \frac{(\sum_{i \in S} y_i)^2}{|S|}.$$

This avoids recomputing residuals from scratch each time.

5.3 Minimal regression tree in Python

Notes: (i) This is a teaching implementation (clarity over speed). (ii) Candidate cutpoints are midpoints between sorted unique feature values.

```
import numpy as np

def rss_from_sums(sum_y, sum_y2, n):
    # RSS = sum(y^2) - (sum(y)^2)/n
    if n <= 0:
        return np.inf
    return sum_y2 - (sum_y * sum_y) / n

def best_split_regression(X, y, idx):
    # Returns (best_j, best_s, best_loss) or (None, None, inf) if no split.
    n, p = X.shape
    yS = y[idx]
    best_loss = np.inf
    best_j = None
    best_s = None

    for j in range(p):
        xj = X[idx, j]
        order = np.argsort(xj)
        x_sorted = xj[order]
        y_sorted = yS[order]

        # Prefix sums to evaluate all split points in O(n)
        psum = np.cumsum(y_sorted)
        psum2 = np.cumsum(y_sorted * y_sorted)
        total_sum = psum[-1]
        total_sum2 = psum2[-1]
        m = len(idx)

        # Candidate splits only where x changes
        for t in range(1, m):
            if x_sorted[t] == x_sorted[t-1]:
                continue
            nL = t
            nR = m - t
```

```

        sumL = psum[t-1]
        sumL2 = psum2[t-1]
        sumR = total_sum - sumL
        sumR2 = total_sum2 - sumL2

        loss = rss_from_sums(sumL, sumL2, nL) + rss_from_sums(sumR, sumR2, nR)
        if loss < best_loss:
            best_loss = loss
            best_j = j
            best_s = 0.5 * (x_sorted[t-1] + x_sorted[t])

    return best_j, best_s, best_loss

def build_tree_regression(X, y, idx=None, depth=0, max_depth=3, min_leaf=5):
    if idx is None:
        idx = np.arange(X.shape[0])

    # Leaf prediction (mean)
    yS = y[idx]
    value = float(np.mean(yS))

    # Stopping criteria
    if depth >= max_depth or len(idx) <= min_leaf:
        return {"is_leaf": True, "value": value}

    j, s, loss = best_split_regression(X, y, idx)
    if j is None or not np.isfinite(loss):
        return {"is_leaf": True, "value": value}

    left_idx = idx[X[idx, j] < s]
    right_idx = idx[X[idx, j] >= s]

    # If a split fails to create two nonempty children, stop
    if len(left_idx) == 0 or len(right_idx) == 0:
        return {"is_leaf": True, "value": value}

    return {
        "is_leaf": False,
        "j": int(j),
        "s": float(s),
        "left": build_tree_regression(X, y, left_idx, depth+1, max_depth, min_leaf),
        "right": build_tree_regression(X, y, right_idx, depth+1, max_depth, min_leaf),
    }

def predict_one(tree, x):
    # Recursive prediction
    if tree["is_leaf"]:
        return tree["value"]
    if x[tree["j"]] < tree["s"]:
        return predict_one(tree["left"], x)
    else:
        return predict_one(tree["right"], x)

def predict(tree, X):
    return np.array([predict_one(tree, X[i]) for i in range(X.shape[0])])

```

5.4 What to emphasize on the board

- **Recursion is the model:** a tree is a function defined by cases; prediction is recursive descent.
- **Greedy split:** each node chooses (j, s) to optimize *local* objective (RSS or impurity).
- **Leaves are simple:** regression leaf = mean; classification leaf = majority class (plus probabilities).
- **Overfitting:** deeper trees reduce training error but can increase test error; prune with α .

5.5 Quick exercises (5–10 minutes)

1. For a node with $y_S = \{2, 2, 8, 8\}$ compute $\bar{y}_S$ and $\text{RSS}(S)$.
2. Suppose a split yields $y_L = \{2, 2\}$ and $y_R = \{8, 8\}$. Compute total RSS after split.
3. In the code, identify the two recursive calls that create the left and right subtrees.
4. Modify `max_depth` and `min_leaf`. Observe training fit vs. generalization on a holdout set.