Decision Trees

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

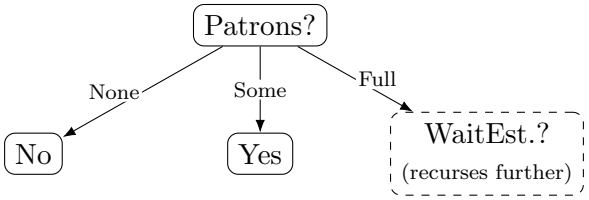
1 What Is a Decision Tree?

- A decision tree maps an input vector x to a decision (discrete → *classification tree*, continuous → *regression tree*) via a sequence of single-feature tests.
- Each internal node tests one feature; each branch is an outcome of that test; each leaf assigns a decision.
- Think of it as *20 questions*: each question should (a) narrow down the answer and (b) can depend on the answers to previous questions.
- To predict: start at the root, follow the branch matching x's feature value, repeat until a leaf, output the leaf's value.

Example (the classic AIMA *restaurant-waiting* problem: predict WillWait from ten features, using 12 training examples $X_1, \ldots, X_{12}$):

	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	**WillWait**
X_1	Y	N	N	Y	Some	\$\$\$	N	Y	French	0–10	**Y**
X_2	Y	N	N	Y	Full	\$	N	N	Thai	30–60	**N**
X_3	N	Y	N	N	Some	\$	N	N	Burger	0–10	**Y**
X_4	Y	N	Y	Y	Full	\$	Y	N	Thai	10–30	**Y**
X_5	Y	N	Y	N	Full	\$\$\$	N	Y	French	>60	**N**
X_6	N	Y	N	Y	Some	\$\$	Y	Y	Italian	0–10	**Y**
X_7	N	Y	N	N	None	\$	Y	N	Burger	0–10	**N**
X_8	N	N	N	Y	Some	\$\$	Y	Y	Thai	0–10	**Y**
X_9	N	Y	Y	N	Full	\$	Y	N	Burger	>60	**N**
X_{10}	Y	Y	Y	Y	Full	\$\$\$	N	Y	Italian	10–30	**N**
X_{11}	N	N	N	N	None	\$	N	N	Thai	0–10	**N**
X_{12}	Y	Y	Y	Y	Full	\$	N	N	Burger	30–60	**Y**

One tree consistent with this data:



- Patrons=None or Some already *perfectly* determines WillWait (pure leaves); only the Full branch needs further questions (WaitEstimate, then Hungry, Alternate, ...).
- Changing the order of tested features gives a *different* (shorter or longer) tree; not every feature must appear (here Type, Price never get tested).

⇒ Learning a tree = choosing *which* feature to test at each node, and *in what order*.

2 Growing a Tree: Notation and the Greedy Algorithm

Training data: N instances, m features, (x_i, y_i), $y_i \in \{1, \ldots, k\}$ (classification). At node q with N_q instances:

$$p_{qk} = \frac{1}{N_q} \sum_{i=1}^{N_q} \mathbb{I}(y_i = k) \qquad \text{(fraction of class } k \text{ at } q) \tag{1}$$

$$class(q) = \arg\max_k p_{qk} \qquad \text{(majority-vote leaf label)} \tag{2}$$

$$Error(q) = \frac{1}{N_q} \sum_{i=1}^{N_q} \mathbb{I}(y_i \neq class(q)) \qquad \text{(training error at } q) \tag{3}$$

- **Why not just minimize $Error(q)$ directly?** Two splits producing $(10, 40)$ / $(40, 10)$ *and* $(50, 20)$ / $(0, 30)$ from a $(50, 50)$ parent both have 20% misclassification error — but the second produces a *pure* node (all one class), which is strictly more useful for further splitting. Misclassification error is blind to purity; we need a different criterion (§3).
- **How many trees are there?** With m binary features, 2^m possible instances, 2^{2^m} possible labelings $\Rightarrow$ for $m = 10$, about 2^{1024} trees. Finding the *smallest* tree consistent with the data is NP-complete $\Rightarrow$ use a greedy heuristic.

Algorithm: Greedy Top-Down Tree Induction (ID3-style)

1. If stopping criterion met (§5): make a leaf, label = majority class (or mean, for regression). Stop.
2. Else: **brute-force** try every feature $\times$ every candidate threshold; pick the (feature, threshold) with best impurity reduction (§3).
3. Split data into children by the winning test.
4. Recurse on each child.

Key point: This is greedy: it commits to a feature/threshold once and never backtracks. Greedy $\neq$ optimal — see §11.

3 Choosing the Best Split: Impurity Measures

Entropy of node q (over k classes):

$$H(q) = -\sum_k p_{qk} \log_2 p_{qk}, \qquad (0 \log_2 0 := 0) \tag{4}$$

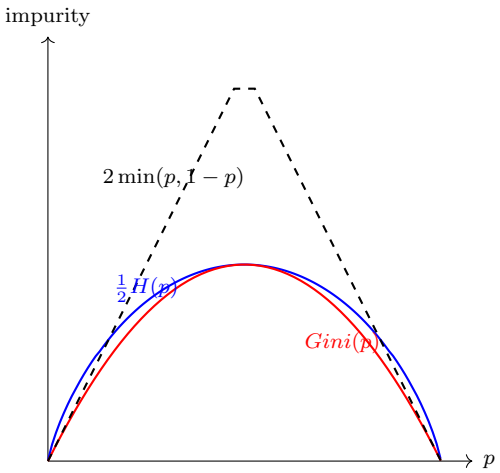
- $H = 0$ iff pure (one class); H is maximized ($= \log_2 k$) at the uniform distribution.
- *Information-theoretic reading*: H is the expected number of bits to encode the class of a random instance under an optimal code, where a message of probability p costs $-\log_2 p$ bits. Rare/surprising outcomes cost more bits; H is the average cost.

Gini index of node q:

$$Gini(q) = \sum_k p_{qk}(1 - p_{qk}) \tag{5}$$

Similar shape to entropy (zero when pure, max at uniform), cheaper to compute (no logs).

Comparison (binary classification, p = fraction of class 1; error curve scaled $\times 2$, entropy scaled $\times \frac{1}{2}$, for visual comparison on $[0, 1]$):



Information gain of splitting q on feature V with $|V|$ branches:

$$IG(q, V) = H(q) - \sum_{i=1}^{|V|} \frac{N_i}{N_q} H(i) \tag{6}$$

Pick the feature maximizing IG (or the Gini-gain analogue).

Worked example 1 (8 points, 2 binary features $X^1, X^2 \to Y$; $H(Y) = .954$):

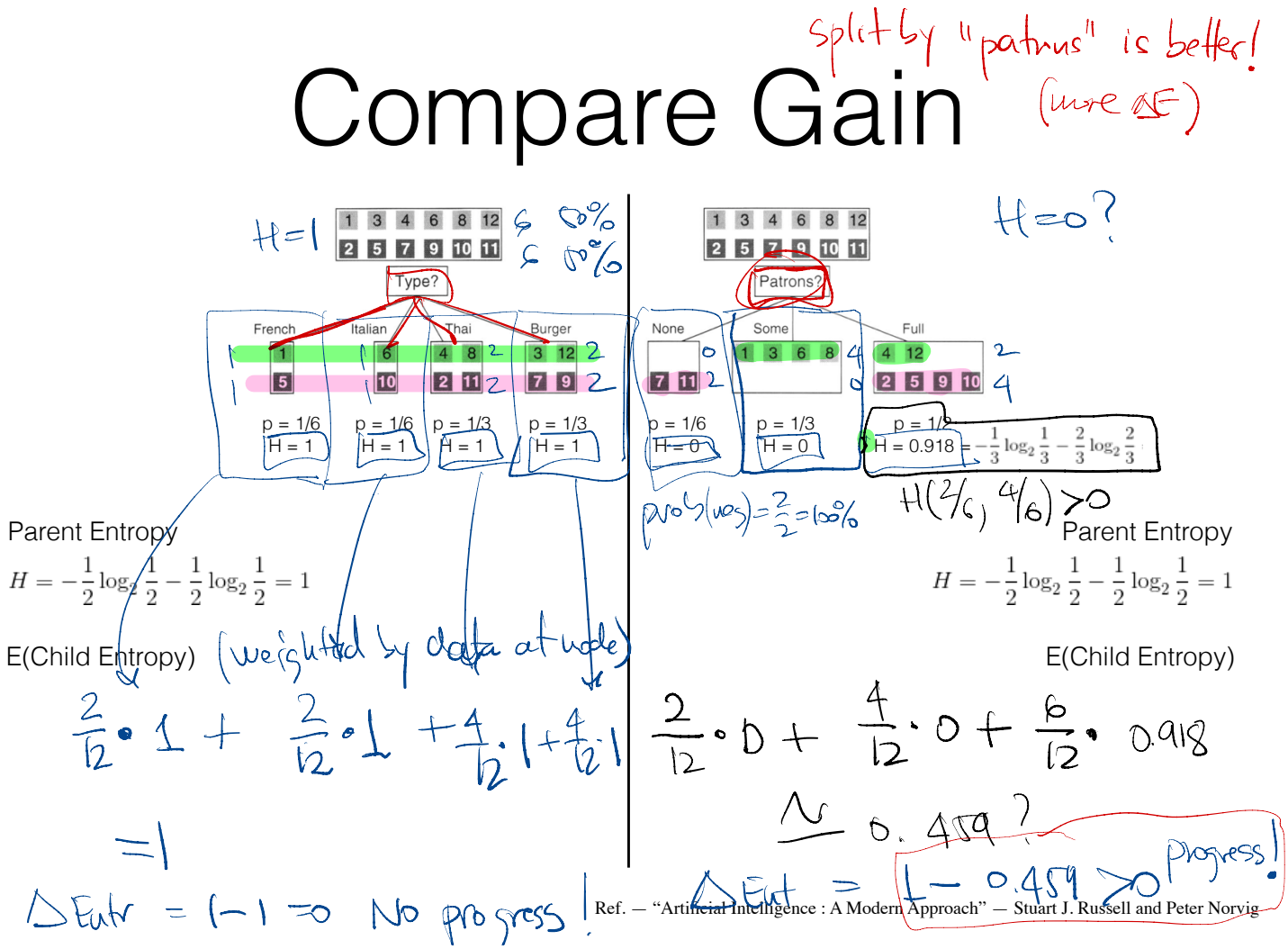
X^1	X^2	Y
T	T	T
T	F	T
T	T	T
T	F	T
F	T	T
F	F	F
F	T	F
F	F	F

$$H(Y|X^1) = \tfrac{1}{2}H(Y|X^1{=}T) + \tfrac{1}{2}H(Y|X^1{=}F) \approx .405 \tag{7}$$
$$IG(X^1) = .954 - .405 = .549 \tag{8}$$

$$H(Y|X^2) = \tfrac{1}{2}H(Y|X^2{=}T) + \tfrac{1}{2}H(Y|X^2{=}F) \approx .905 \tag{9}$$
$$IG(X^2) = .954 - .905 = .049 \tag{10}$$

X^1 is the far better split ($.549 \gg .049$).

Worked example 2 (same 12-example restaurant data as §1; comparing two candidate root splits side by side, since that's the entire point of comparing gain):



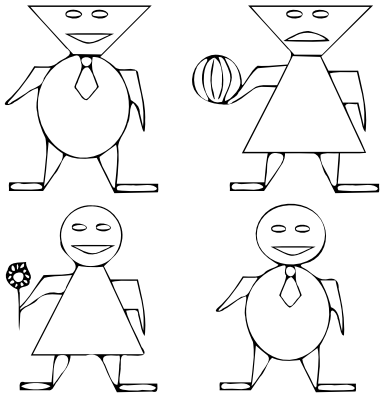
- Split on TYPE (4-way): every child is a 50/50 mix ⇒ each child has $H = 1$, same as the parent ⇒ weighted child entropy $= 1$ ⇒ $\Delta H = 1 - 1 = 0$. **No progress** — TYPE tells us nothing.
- Split on PATRONS (3-way): None and Some children are already pure ($H = 0$); only Full is mixed ($H = 0.918$) ⇒ weighted child entropy $= \frac{6}{12}(0.918) \approx 0.459$ ⇒ $\Delta H = 1 - 0.459 > 0$. **Progress** — and indeed PATRONS is the better root split (matches §1's tree).

Worked example 3 (a different domain, to see the full arithmetic once with a categorical feature set: classify cartoon robots as *ally* or *enemy* from shape attributes HEAD, BODY, SMILE, NECK, HOLDS):

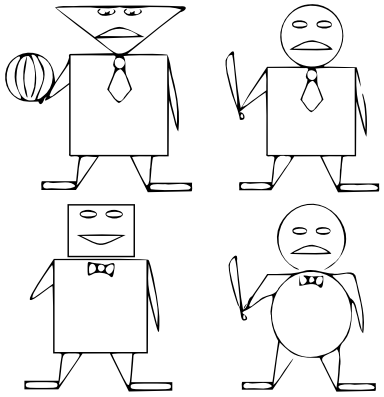
A computer game

Example 2:
Some robots changed their attitudes:

Allies



Enemies



No obvious simple rule.
How to build a decision tree discriminating the 2 robot classes?

No single attribute is perfect (splits everyone into pure classes) or useless (splits everyone into the same mix as the parent) — the 8 examples and per-value class counts:

Attribute importance

head	body	smile	neck	holds	class
triangle	circle	yes	tie	nothing	ally
triangle	triangle	no	nothing	ball	ally
circle	triangle	yes	nothing	flower	ally
circle	circle	yes	tie	nothing	ally
triangle	square	no	tie	ball	enemy
circle	square	no	tie	sword	enemy
square	square	yes	bow	nothing	enemy
circle	circle	no	bow	sword	enemy
triangle: 2:1	triangle: 2:0	yes: 3:1	tie: 2:2	ball: 1:1	
circle: 2:2	circle: 2:1	no: 1:3	bow: 0:2	sword: 0:2	
square: 0:1	square: 0:3		nothing: 2:0	flower: 1:0	
				nothing: 2:1	

A perfect attribute divides the examples into sets each of which contain only a single class. (Do you remember the simply created perfect attribute from Example 1?)

A useless attribute divides the examples into sets each of which contains the same distribution of classes as the set before splitting.

None of the above attributes is perfect or useless. Some are more useful than others.

Full gain calculation for all five attributes (S = all 8 examples, $H(C, S) = 1$ since 4 allies/4 enemies):

head:
$p(S_{\text{head=tri}}) = \frac{3}{8}$; $H(C, S_{\text{head=tri}}) = H_B\left(\frac{2}{2+1}\right) = 0.92$
$p(S_{\text{head=cir}}) = \frac{4}{8}$; $H(C, S_{\text{head=cir}}) = H_B\left(\frac{2}{2+2}\right) = 1$
$p(S_{\text{head=sq}}) = \frac{1}{8}$; $H(C, S_{\text{head=sq}}) = H_B\left(\frac{0}{0+1}\right) = 0$
$H(C, S, head) = \frac{3}{8} \cdot 0.92 + \frac{4}{8} \cdot 1 + \frac{1}{8} \cdot 0 = 0.84$
$Gain(head, S) = 1 - 0.84 = 0.16$

body:
$p(S_{\text{body=tri}}) = \frac{2}{8}$; $H(C, S_{\text{body=tri}}) = H_B\left(\frac{2}{2+0}\right) = 0$
$p(S_{\text{body=cir}}) = \frac{3}{8}$; $H(C, S_{\text{body=cir}}) = H_B\left(\frac{2}{2+1}\right) = 0.92$
$p(S_{\text{body=sq}}) = \frac{3}{8}$; $H(C, S_{\text{body=sq}}) = H_B\left(\frac{0}{0+3}\right) = 0$
$H(C, S, body) = \frac{2}{8} \cdot 0 + \frac{3}{8} \cdot 0.92 + \frac{3}{8} \cdot 0 = 0.35$
$Gain(body, S) = 1 - 0.35 = 0.65$

smile:
$p(S_{\text{smile=yes}}) = \frac{4}{8}$; $H(C, S_{\text{yes}}) = H_B\left(\frac{3}{3+1}\right) = 0.81$
$p(S_{\text{smile=no}}) = \frac{4}{8}$; $H(C, S_{\text{no}}) = H_B\left(\frac{1}{1+3}\right) = 0.81$
$H(C, S, smile) = \frac{4}{8} \cdot 0.81 + \frac{4}{8} \cdot 0.81 + \frac{3}{8} \cdot 0 = 0.81$
$Gain(smile, S) = 1 - 0.81 = 0.19$

neck:
$p(S_{\text{neck=tie}}) = \frac{4}{8}$; $H(C, S_{\text{neck=tie}}) = H_B\left(\frac{2}{2+2}\right) = 1$
$p(S_{\text{neck=bow}}) = \frac{2}{8}$; $H(C, S_{\text{neck=bow}}) = H_B\left(\frac{0}{0+2}\right) = 0$
$p(S_{\text{neck=no}}) = \frac{2}{8}$; $H(C, S_{\text{neck=no}}) = H_B\left(\frac{2}{2+0}\right) = 0$
$H(C, S, neck) = \frac{4}{8} \cdot 1 + \frac{2}{8} \cdot 0 + \frac{2}{8} \cdot 0 = 0.5$
$Gain(neck, S) = 1 - 0.5 = 0.5$

holds:
$p(S_{\text{holds=ball}}) = \frac{2}{8}$; $H(C, S_{\text{holds=ball}}) = H_B\left(\frac{1}{1+1}\right) = 1$
$p(S_{\text{holds=swo}}) = \frac{2}{8}$; $H(C, S_{\text{holds=swo}}) = H_B\left(\frac{0}{0+2}\right) = 0$
$p(S_{\text{holds=flo}}) = \frac{1}{8}$; $H(C, S_{\text{holds=flo}}) = H_B\left(\frac{1}{1+0}\right) = 0$
$p(S_{\text{holds=no}}) = \frac{3}{8}$; $H(C, S_{\text{holds=no}}) = H_B\left(\frac{2}{2+1}\right) = 0.92$
$H(C, S, holds) = \frac{2}{8} \cdot 1 + \frac{2}{8} \cdot 0 + \frac{1}{8} \cdot 0 + \frac{3}{8} \cdot 0.92 = 0.6$
$Gain(holds, S) = 1 - 0.6 = 0.4$

The **body** attribute

- ✔ brings us the largest information gain, thus
- ✔ it shall be chosen for the first test in the tree!

BODY wins ($Gain = 0.65$, largest of the five) and is chosen for the root test.

Remark. **Gain is biased toward high-arity features.** A feature with a unique value per instance (e.g. a `Date` column) gets $IG = H(q)$ (max possible) but generalizes to nothing. Fix (Quinlan's C4.5):

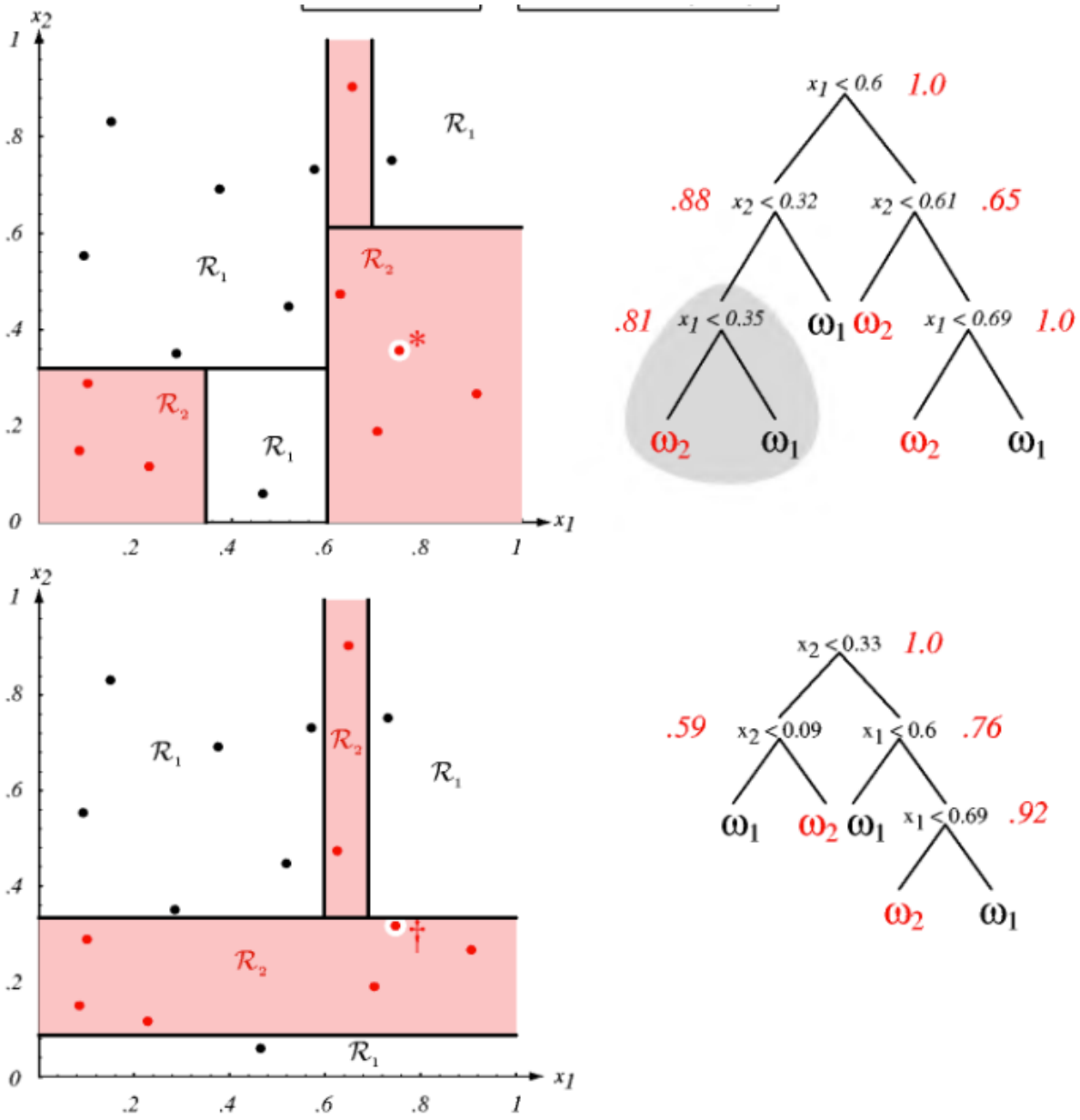
$$Gain\ Ratio(q, V) = \frac{IG(q, V)}{SplitInfo(q, V)} \tag{11}$$

$$SplitInfo(q, V) = -\sum_i \frac{N_i}{N_q} \log_2 \frac{N_i}{N_q} \tag{12}$$

$SplitInfo$ penalizes many small branches. **Our HW1 tree uses raw IG/variance-reduction, not gain ratio** — a known simplification, fine for numeric-feature binary splits (where arity is always 2) but worth knowing about.

4 Splitting Different Feature Types

- **Ordinal** (e.g. Price $\in \{\$, \$\$, \$\$\$\}$): pick threshold θ, split $x < \theta$ vs. $x \geq \theta$.
- **Numerical/continuous**: sort the feature's values; **candidate thresholds = midpoints between consecutive *distinct* sorted values**, $\frac{x_{(j)} + x_{(j+1)}}{2}$; evaluate IG at each; keep the best. (No need to test more than $N - 1$ candidates — this is exactly `candidate_thresholds` in HW1.)
- **Nominal/categorical** with v values: a binary split partitions the v values into 2 non-empty groups $\Rightarrow 2^{v-1} - 1$ possible partitions to search (exhaustively, or replace by v binary dummy features).
- **Binary vs. multiway**: prefer binary splits — any k-ary split is equivalent to a sequence of binary splits, and multiway splits fragment data faster, leaving less for deeper levels.



A univariate tree only ever cuts axis-aligned rectangles out of feature space — each root-to-leaf path is a conjunction of $x_j \lessgtr \theta$ tests.

5 When to Stop — and a Zero-Gain Subtlety

Stop splitting a node when:

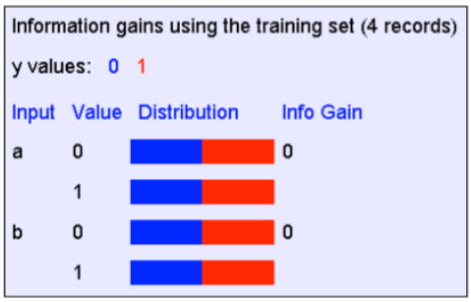
- all instances have the same label ($H(q) = 0$), or
- no feature improves impurity ($IG = 0$ for every candidate split, i.e. $H(Y|X) = H(Y)$).

Remark. Stopping the instant $IG = 0$ can be short-sighted: a feature useless *by itself* may become useful *combined with another* (classic XOR-like case). Below, neither half of the top split is pure, but splitting again below it (right) achieves perfect separation that stopping early (left) would have missed.

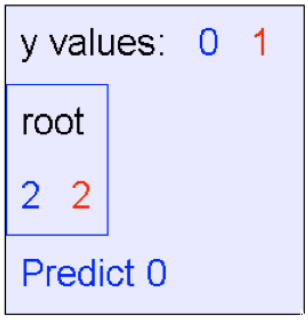
a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

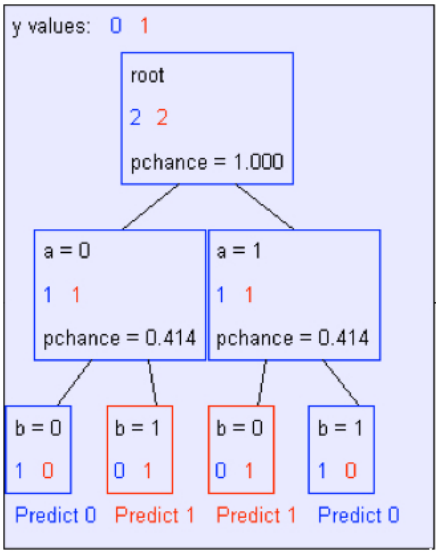
y = a XOR b

The information gains:



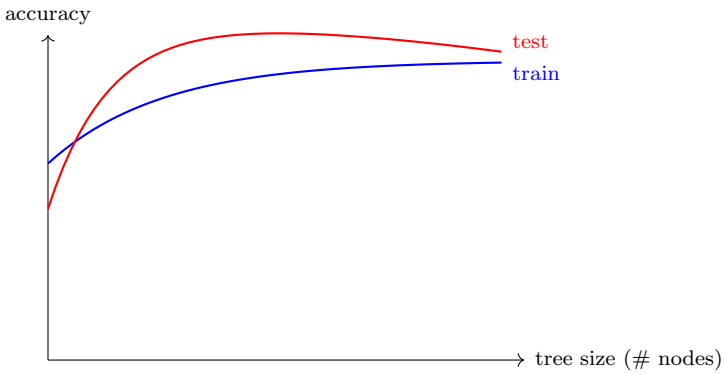
The resulting decision tree:





6 Overfitting

Definition. h overfits if $\exists h'$ with $error_{train}(h) < error_{train}(h')$ but $error_{true}(h) > error_{true}(h')$ — h looks better only because it memorized the training set.



Training accuracy climbs monotonically; test accuracy peaks early, then *drops* as the tree memorizes noise — the gap after the peak is overfitting.

Key point: A fully-grown tree (splits until pure) drives training error to zero *provided no two training points have identical features but different labels* (if they do, that leaf can never be pure, no matter how deep the tree goes) — and even then, it's exactly the high-variance regime above. Fix: either stop growing early, or grow fully and cut back afterward — §7.

7 Pruning

Two families: **pre-pruning** (stop growing early) and **post-pruning** (grow fully, then cut back). Post-pruning almost always wins in practice — a split that looks useless in isolation can still enable a great split further down (the same horizon effect as §5's zero-gain subtlety), so pre-pruning can stop too soon. Post-pruning grows the full tree first, so it never misses that interaction; it only pays extra compute.

Pre-pruning (early stopping) — cap, before growing:

- max depth; max # leaves; min # instances per leaf/split; min impurity decrease required to accept a split.

- Cheap (never grows the parts you'd cut anyway) but short-sighted, and the right cap is dataset-dependent with no universal setting — exactly why HW1 Problem 2 asks you to sweep a few depths and compare train/test error. This is the only form of pruning HW1's tree implements.

Reduced-error pruning (Quinlan, 1987) — needs a held-out validation set, separate from both train and test:

Algorithm: Reduced-Error Pruning

---

1. Grow the full tree T on training data (no early stopping).

2. For each internal node t, greedily: let $T' = T$ with the subtree at t replaced by a single majority-class leaf.

3. If $error(T', D_{val}) \leq error(T, D_{val})$: accept the pruned version, $T \leftarrow T'$.

4. Repeat (re-scanning remaining internal nodes) until no single pruning helps. Result: smallest subtree of the fully-grown T that doesn't hurt validation accuracy.

Simple and effective, but wasteful of data (an entire validation split is set aside purely for pruning decisions) and its greedy node-by-node order can miss a jointly-optimal prune.

Cost-complexity ("weakest-link") pruning (Breiman, Friedman, Olshen & Stone, *CART*, 1984) — the standard in modern libraries (this is what `sklearn`'s `ccp_alpha` controls). For a subtree T with leaves $\widetilde{T}$, define a penalized cost

$$R_\alpha(T) = R(T) + \alpha\,|\widetilde{T}|, \qquad R(T) = \sum_{t \in \widetilde{T}} R(t) \tag{13}$$

where $R(t)$ is the training error (or SSE, for regression) at leaf t, $|\widetilde{T}|$ is the number of leaves, and $\alpha \geq 0$ is a complexity penalty per leaf — the same bias-variance knob as L_1/L_2 regularization, just for tree size instead of weight magnitude. $\alpha = 0$ recovers the fully-grown tree ($R(T)$ alone, minimized by splitting everything); $\alpha \to \infty$ collapses to the root.

Algorithm: Weakest-Link Pruning (builds a sequence, then picks one)

1. Grow the full tree T_0.
2. For each internal node t, let T_t = the subtree rooted at t, and define its *effective* α

$$g(t) = \frac{R(t) - R(T_t)}{|\widetilde{T}_t| - 1} \tag{14}$$

   (the break-even penalty at which collapsing T_t to a leaf stops costing more than it saves).
3. Prune the node $t^* = \arg\min_t g(t)$ (the "weakest link"), giving T_1; record $\alpha_1 = g(t^*)$.
4. Repeat on $T_1, T_2, \ldots$ until only the root remains — this produces a finite *nested* sequence $T_0 \supset T_1 \supset \cdots \supset \{root\}$ with increasing $\alpha_0 < \alpha_1 < \cdots$.
5. Pick the final tree by K-fold cross-validation over this sequence (evaluate each T_i on held-out folds, pick the α — equivalently the T_i — with lowest CV error; the "1-SE rule" picks the *smallest* tree within one std. error of the CV minimum, favoring simplicity).

Unlike reduced-error pruning, this needs no dedicated validation split held out in advance — the sequence is built once from training data alone, and only the final α selection uses cross-validation.

Rule post-pruning (Quinlan's C4.5, 1993): convert each root-to-leaf path into an IF-THEN rule; drop preconditions from each rule independently if doing so doesn't hurt validation accuracy; sort surviving rules by accuracy for use. More flexible than tree-pruning since a precondition can be dropped from one rule without affecting sibling rules that shared the same ancestor split.

Key point: All post-pruning methods share the same shape: grow greedily and fully (cheap, and never misses an interaction), then remove complexity using a criterion the greedy growth phase didn't have access to (a validation set, cross-validation, or an explicit penalty). Growing and pruning are solving two different problems — don't expect one greedy pass to do both.

8 Missing Values and Two Kinds of Cost

Missing feature values — options, roughly cheapest to most sophisticated:

- drop incomplete instances (if data is plentiful); drop the feature if $\gtrsim 70\%$ missing.
- impute with the feature's overall mean, or its mean *within the instance's class.*
- if "missing" is itself meaningful (e.g. a medical test not administered), make it its own category.
- at test/split time: send the instance down *every* branch, weighted by the fraction of training instances that went each way (C4.5's fractional-instance approach); classify by combining the weighted leaf votes.

Misclassification cost (not all errors are equal — e.g. false-negative cancer diagnosis $\gg$ false-positive): a $k \times k$ loss matrix L_{ij} = cost of predicting j when truth is i ($L_{ii} = 0$). Cost-sensitive Gini:

$$Gini_L(q) = \sum_{k \neq k'} L_{kk'}\, p_{qk}\, p_{qk'} \tag{15}$$

Attribute-test cost (a different notion — some features are expensive/slow to *measure*, e.g. a $150 blood test): prefer cheap, high-gain features. Two proposed scores (larger = better):

$$\text{(Tan \& Schlimmer)} \qquad \frac{IG(q,V)^2}{Cost(V)} \tag{16}$$

$$\text{(Núñez)} \qquad \frac{2^{IG(q,V)}-1}{(Cost(V)+1)^w}, \quad w \in [0,1] \text{ tunes cost-sensitivity} \tag{17}$$

9 Regression Trees

Same recursion, different impurity: for node m with points χ_m, predict the mean and measure spread by variance/SSE:

$$g_m = \frac{1}{|\chi_m|}\sum_{t\in\chi_m} y_t \qquad \text{(predicted value at node } m\text{)} \tag{18}$$

$$E_m = \frac{1}{|\chi_m|}\sum_{t\in\chi_m} (y_t - g_m)^2 \qquad \text{(mean squared error at } m\text{)} \tag{19}$$

For a candidate split of χ_m into $\chi_{m1}, \ldots, \chi_{mk}$ (by feature X):

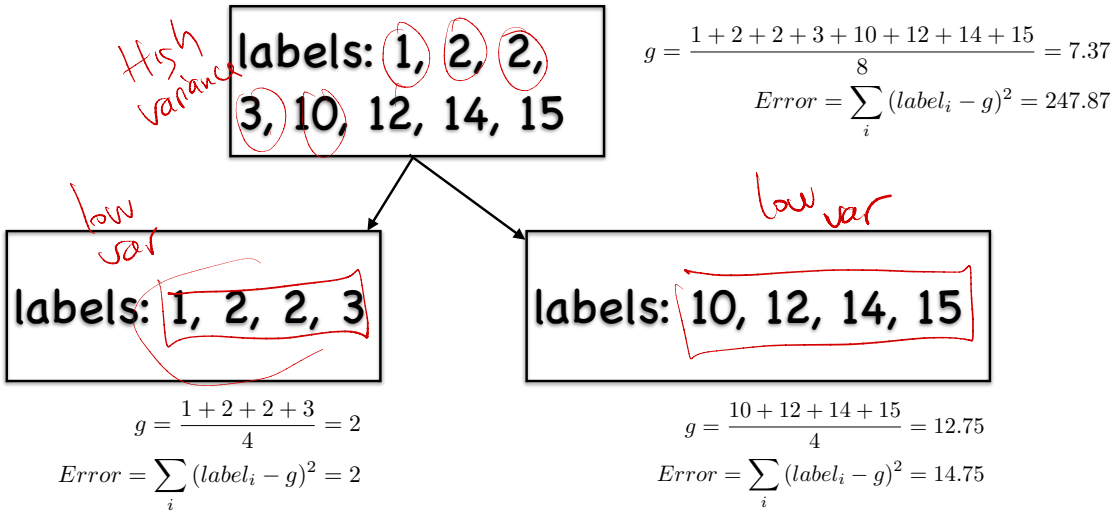
$$g_{mj} = \frac{1}{|\chi_{mj}|}\sum_{t\in\chi_{mj}} y_t \tag{20}$$

$$E'_m(X) = \frac{1}{|\chi_m|}\sum_j \sum_{t\in\chi_{mj}} (y_t - g_{mj})^2 \tag{21}$$

Choose X (and threshold) minimizing $E'_m(X)$, i.e. maximizing the drop $E_m - E'_m(X)$ — **variance reduction is the regression analogue of information gain.**

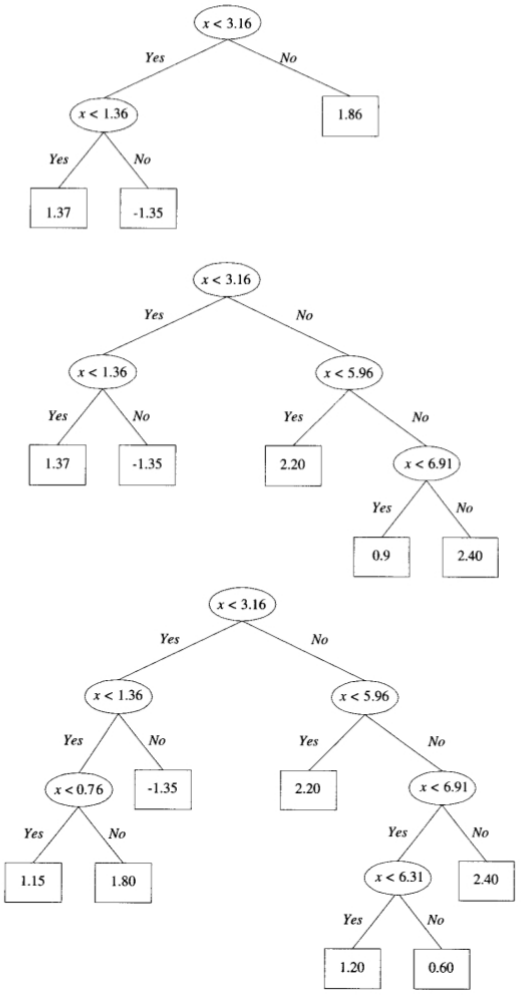
Worked example: labels $\{1, 2, 2, 3, 10, 12, 14, 15\}$.

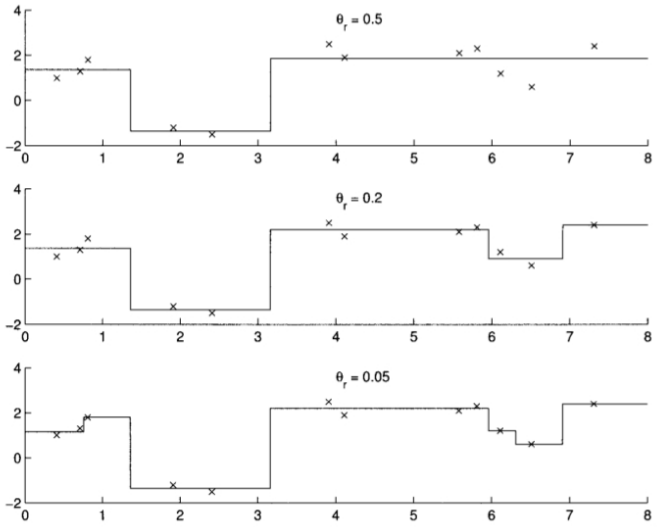
Regression Tree



- choose a split criteria to minimize the weighted or total error at children nodes
  - in the example total error after the split is 14.75 + 2=16.75

Parent: $g = 7.37$, $SSE = 247.87$ (high — widely spread). Split into $\{1, 2, 2, 3\}$ ($g = 2$, $SSE = 2$) and $\{10, 12, 14, 15\}$ ($g = 12.75$, $SSE = 14.75$): total post-split $SSE = 16.75 \ll 247.87$ — a huge variance reduction, so this split is excellent.





A different (real, car-price) regression tree and its resulting step-function fit — deeper trees fit finer step functions, same overfitting trade-off as classification.

	Classification tree	**Regression tree**
impurity	entropy / Gini	variance / SSE
splitting criterion	Δentropy = information gain	ΔSSE
leaf prediction	majority vote	mean

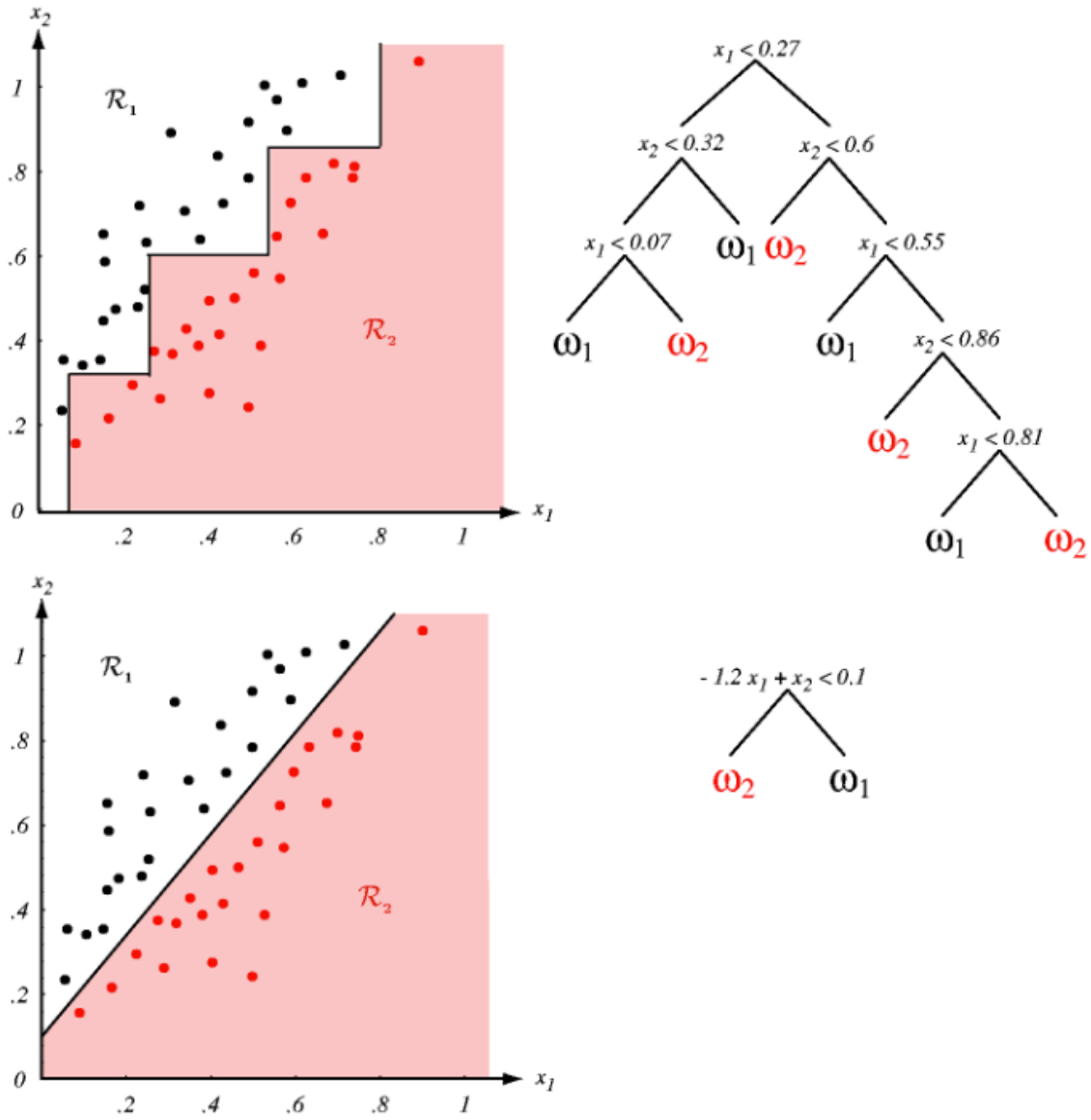
Key point: Classification and regression trees are *one algorithm* with a swappable impurity function. HW1's `DecisionTree` class implements exactly this: `mode="classification"` $\rightarrow$ entropy, `mode="regression"` $\rightarrow$ variance, identical recursion otherwise.

10 Multivariate (Oblique) Trees

Univariate splits only cut axis-aligned rectangles (§5's figure). If the true boundary is diagonal, a univariate tree needs many splits to approximate it; a **multivariate/oblique** split tests a linear combination of features:

$$w_1 x_1 + w_2 x_2 + \cdots + w_m x_m + w_0 > 0 \tag{22}$$

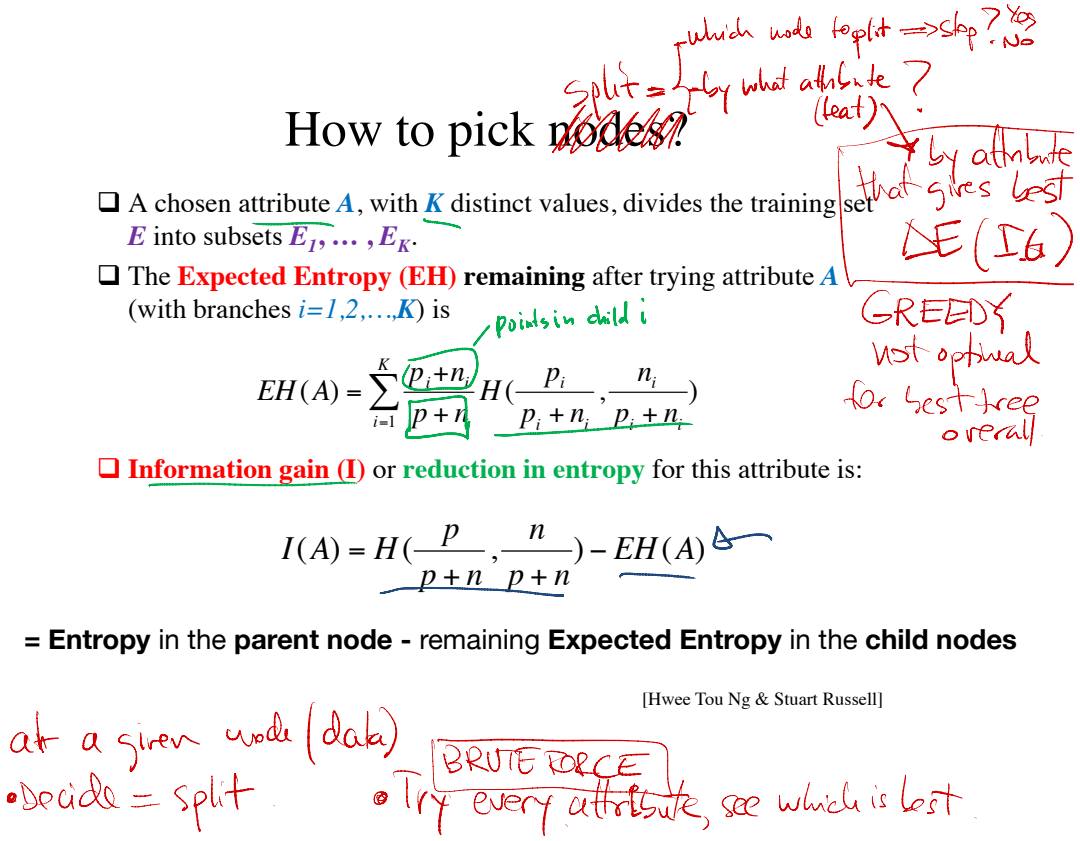
e.g. a single split $2x_1 + 1.5x_2 < 0.5$ can perfectly separate data that would need many axis-aligned splits.



Target concept is linear in (x_1, x_2): univariate tree (top) needs a staircase of splits to approximate it; multivariate tree (bottom) solves it in one split.

Remark. Power has a price: finding the best oblique split is no longer brute-forceable — $2^d \binom{N}{d}$ candidate hyperplanes for d-dimensional splits. (We'll see efficient ways to find good hyperplanes when we cover linear classifiers/SVMs later in the course.)

11 Greedy ≠ Optimal, and Occam's Razor



- ID3's search is **greedy hill-climbing**: pick the best feature *now*, never backtrack. This can miss the globally best tree (a locally mediocre split might set up a much better tree two levels down).
- Statistically-based choices (impurity over many instances, not exact consistency) give some robustness to noisy data, unlike a search that insists on perfectly fitting every instance.
- **Occam's Razor** ("prefer the shortest hypothesis consistent with data"): fewer short trees exist, so one that fits by pure chance is less likely ⇒ mild justification for preferring small trees.
- *Counter-argument*: "small" is arbitrary — e.g. the (tiny) class of "trees with a prime number of nodes using only features starting with Z" is small by fiat, not by any principled measure. Occam's Razor is a heuristic preference, not a theorem.

12 Continuity: Trees → Boosting

- HW1 Problem 2 builds this exact recursive splitter (entropy for Spambase, variance for Housing) as a **reusable `fit()`/`predict()` module**.
- HW1 Problem 3 reuses it unchanged as the *weak learner* inside residual boosting: fit a shallow tree to the residual, subtract its prediction, repeat — the entire ensemble is a four-line loop around the class from §13:

```
def fit(self, X, y):                          # BoostedRegressionTrees.fit
    residual = y.copy()
    for _ in range(self.n_rounds):
        tree = DecisionTree(mode="regression", max_depth=self.tree_depth)
        tree.fit(X, residual)
        residual = residual - tree.predict(X)  # what's left for the next tree to explain
        self.trees.append(tree)

def predict(self, X):
    return sum(tree.predict(X) for tree in self.trees)   # sum of all rounds' outputs
```

Note what did *not* change from a single tree: no re-weighting of examples (that's AdaBoost, a different boosting flavor), no learning rate here (Problem 5's Newton-boosting variant adds one). Just: fit the errors, add it in, repeat.

- HW1 Problem 5 (optional) generalizes the same recursion to a *second-order* (Newton/XGBoost-style) tree: same shape, but splits are scored by a gradient/Hessian structure score instead of entropy/variance, and leaves get a closed-form optimal weight instead of a majority vote/mean.

⇒ One well-built tree module underpins three HW1 problems and reappears in HW2's boosting. Get §13 right once.

13 Implementation Notes (HW1)

Design. One class, one impurity function swapped by mode:

- `fit(X,y)` → recursively grows a tree of `TreeNode`s; `predict(X)` → walks each row root-to-leaf.
- Node stores: `feature`, `threshold`, `left`, `right` (internal) *or* `value` (leaf). `is_leaf()` ≡ `value is not None`.

Candidate thresholds & complexity. Per node, per feature: sort → dedupe → take midpoints of consecutive values (§3) → evaluate impurity at each. Naively, this is $O(n \log n)$ per feature per node (re-sorting every time) × m features × (up to) n candidate thresholds each costing $O(n)$ to score ⇒ the brute-force split search at one node is $O(mn^2)$. Fine for HW1's dataset sizes; a real implementation would pre-sort each column once and maintain running class counts while sweeping thresholds left-to-right ($O(mn \log n)$ total) — a good optimization exercise, not required here.

Generic recursive split search (both classification and regression)

1. Base case: depth limit reached, or node is pure, or too few points → return a leaf (majority class / mean).
2. Else: for every feature, every candidate threshold → compute weighted-child impurity → track the best (feature, threshold, gain).
3. If best gain ≤ 0: return a leaf anyway (no split helps).
4. Else: partition on the winning test, recurse left and right, attach as children.

In actual HW1 code (`mode="classification"` ⇒ `self.impurity=entropy`; `"regression"` ⇒ `variance` — same recursion either way, §2's "one class, one impurity function" point made concrete):

```
def entropy(y):
    _, counts = np.unique(y, return_counts=True)
    p = counts / len(y)
    return -np.sum(p * np.log2(p + 1e-12))          # +eps avoids log2(0)

def variance(y):
    return np.mean((y - np.mean(y)) ** 2) if len(y) else 0.0

def candidate_thresholds(column):
    values = np.unique(column)                       # sorted, deduped
    return (values[:-1] + values[1:]) / 2.0          # midpoints, Sec.4

def _grow(self, X, y, depth):
    n, m = X.shape
    if depth >= self.max_depth or len(np.unique(y)) == 1 or n < self.min_samples_split:
        return TreeNode(value=self._leaf_value(y))    # stopping criterion, Sec.5

    parent_impurity = self.impurity(y)
    best_gain, best_feature, best_threshold = 0.0, None, None
    for feature in range(m):
        for threshold in candidate_thresholds(X[:, feature]):
            left_mask = X[:, feature] <= threshold
            if left_mask.sum() == 0 or (~left_mask).sum() == 0:
                continue                              # empty branch, skip
            y_left, y_right = y[left_mask], y[~left_mask]
            weighted_child_impurity = (len(y_left)/n * self.impurity(y_left)
                                       + len(y_right)/n * self.impurity(y_right))
            gain = parent_impurity - weighted_child_impurity
            if gain > best_gain:
                best_gain, best_feature, best_threshold = gain, feature, threshold

    if best_feature is None:                          # no split helps -> leaf
        return TreeNode(value=self._leaf_value(y))
    left_mask = X[:, best_feature] <= best_threshold
    left  = self._grow(X[left_mask],  y[left_mask],  depth + 1)
    right = self._grow(X[~left_mask], y[~left_mask], depth + 1)
    return TreeNode(feature=best_feature, threshold=best_threshold, left=left, right=right)
```

Line-by-line, this *is* §3–§5: `parent_impurity - weighted_child_impurity` is Eq. 6 (with `entropy`) or the variance-reduction analogue (with `variance`); the `depth`/`unique(y)`/`min_samples_split` check is §5's stopping rule; `best_feature is None` is exactly the zero-gain subtlety's "no candidate helps, stop anyway."

Sanity-checking your tree. Before trusting train/test numbers, check structure, not just accuracy:

- On a *tiny* hand-checkable dataset (e.g. Worked example 1 or 3 above), verify your tree's root split matches the Gain you computed by hand.
- Fit `sklearn`'s classification or regression tree at depth 1:

  `DecisionTreeClassifier(criterion='entropy', max_depth=1)`

  `DecisionTreeRegressor(criterion='squared_error', max_depth=1)`

  on the *same* data — HW1 Problem 2(a) already asks for this library baseline — and check it picks the same `(feature, threshold)` at the root. If it doesn't, the bug is almost always the direction-mismatch or empty-branch issue below, not the impurity formula itself (those tend to be visibly wrong on the tiny example first).
- Two accuracy numbers being "close but not identical" to `sklearn` at `max_depth=1` is a red flag, not noise — with one split, both should find the *exact* same optimum (it's a small discrete search, not an iterative optimizer with randomness).

Common bugs (all silent — wrong answer, not a crash):

- **Direction mismatch**: the split search must partition with *exactly* the same comparison ('$\leq \theta$' vs. '$< \theta$') that `predict`'s tree-walk uses. A mismatch silently mis-routes boundary points.

- $\log_2(0)$: entropy needs $0 \log_2 0 := 0$ (add a tiny ϵ inside the log, or special-case it) — otherwise NaNs propagate silently through the gain computation.
- **Empty-branch splits**: a candidate threshold that sends *all* points to one side has undefined child impurity on the other side — skip it, don't score it.
- **Depth vs. recursion**: an unbounded `max_depth` on a large, noisy dataset can recurse very deep (Python's default recursion limit, or just a fully-memorized, useless tree) — always cap depth or min-samples-per-split.

14 Optional: Why Can Information Gain Not Exceed 1 Bit? (Proof Sketch)

For a **binary** split on feature X (2 branches) with binary/multi-class label Y: $IG(q, X) = H(Y) - H(Y|X)$ is exactly the **mutual information** $I(X, Y)$:

$$I(X, Y) = \sum_{x,y} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \tag{23}$$

$$H(Y|X) = \sum_{x,y} p(x, y) \log \frac{p(x)}{p(x, y)} \tag{24}$$

- Expand $I(X, Y)$ using $\sum_x p(x, y) = p(y)$ (marginalizing) $\Rightarrow$ recover $I(X, Y) = H(Y) - H(Y|X)$.
- **Symmetry**: the same algebra gives $I(X, Y) = H(X) - H(X|Y)$ — entropy reduction in Y from knowing X equals entropy reduction in X from knowing Y.
- Now let X be a *binary* split indicator. Use the symmetry: bound $H(X) - H(X|Y)$ instead of $H(Y) - H(Y|X)$ directly — and $H(X) \leq 1$ bit always, since X is binary.

Key point: Finishing the bound from here (using $H(X) \leq 1$ and $H(X|Y) \geq 0$) is HW1's optional Problem 11 ("Entropy Decrease is Bounded") — this identity and the symmetry trick are exactly the hint given there.