Trees and ensembles

Hicham Janati Fall 2026

Every model in this course so far carried a structural commitment: a Gaussian for the data (chapter 1), Gaussian components (chapter 2), a linear conditional mean (chapter 4), linear scores under every loss (chapter 5). This chapter goes to the far end of the assumptions spectrum. Decision trees abandon the global distributional and functional assumptions of the previous chapters and, true to the course's fil rouge, that freedom comes at a cost in variance. The chapter builds the tree from scratch, measures its instability, then develops the two ways of combining trees that are now standard on tabular data: averaging (bagging, random forests) and sequential correction (boosting, up to XGBoost and LightGBM). It closes on a real housing benchmark that compares all the course's predictors on the same data.

Throughout, the course conventions hold: bold letters denote vectors and matrices, plain italic letters denote scalars.

1 Decision trees

1.1 A classification tree, by hand

Before any algorithm, a decision tree is a codified way of reasoning that humans use spontaneously. A credit officer might work like this: *if the applicant's debt ratio exceeds 35%, refuse; otherwise, if income is below 30k, send to manual review; otherwise, approve.* Figure 1 draws this policy twice. On the left, as a flowchart: each internal node asks a yes/no question about *one* feature, each terminal node (a *leaf*) carries a decision. On the right, as geometry: since every question is a comparison of one coordinate to a threshold, following the branches carves the feature space into *axis-aligned rectangles*, one per leaf.

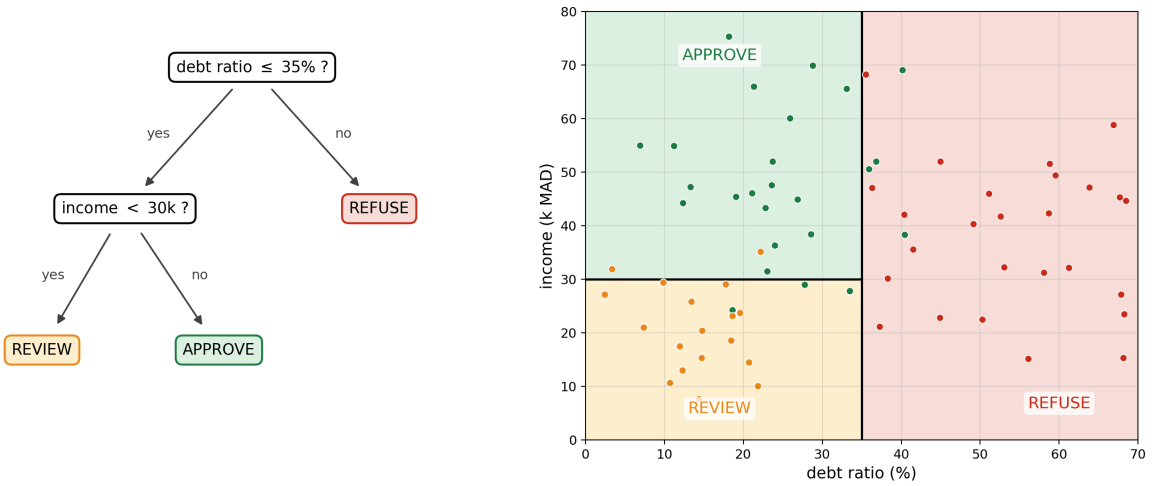


Figure 1: The same decision policy as a tree (left) and as the partition of feature space it induces (right): one rectangle per leaf. The dots are training applicants colored by their true outcome. The partition is not pure (nine of the 69 points land in the wrong rectangle), but the majority vote in every leaf recovers the right decision.

The *model* is the partition plus one decision per rectangle; for classification, each leaf carries a class, or better the class proportions among the training points that fall in it. The *learning problem* is to choose the questions (which feature, which threshold, in which order) from data. And the two virtues that made trees popular are already visible: no assumption anywhere (no Gaussianity, no linearity, features on arbitrary scales), and complete readability, since the path to a leaf *is* the explanation of the decision.

1.2 Regression trees

Nothing in the picture requires a discrete target. For a continuous response, the same partition machinery applies, and each leaf carries a number instead of a class: the *average* of the training responses that fall in it. Figure 2 develops the construction one level deeper than figure 1, on a rent-prediction task with two features. Three questions (on the surface, then the distance to the center, then the surface again) carve the plane into five regions $R_1, \dots, R_5$, and the number attached to each region is the average rent of the training apartments that fall in it: the region R_4 of large central apartments collects the highest rents, so its eight members average to 10.3 k MAD, and every new apartment routed there is predicted at 10.3. Note the two occurrences of *surface*: the same feature may be questioned again at a deeper node, with a different threshold on a different subset of the data.

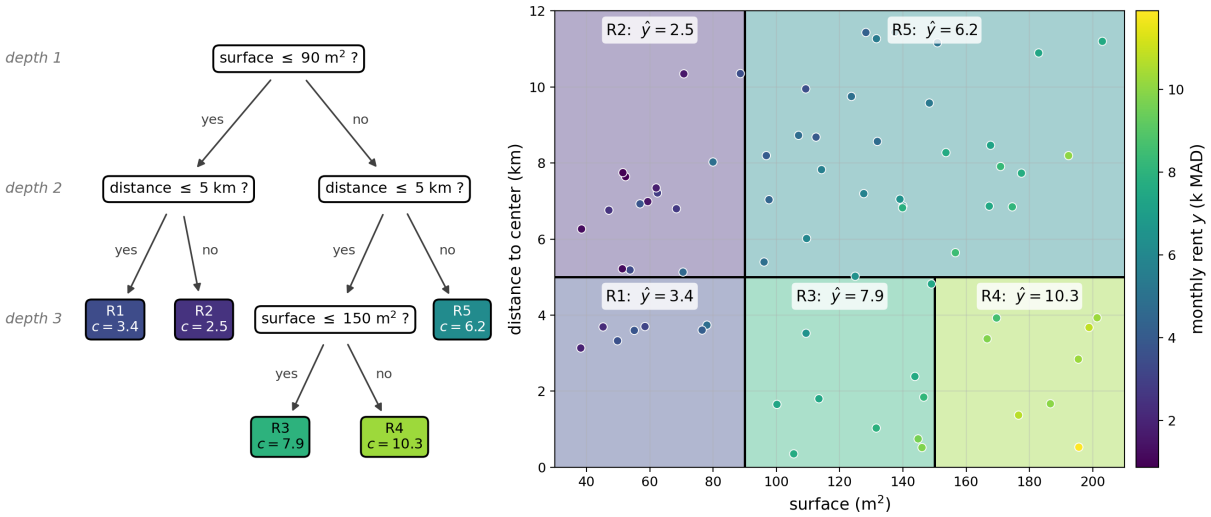


Figure 2: A regression tree of depth 3 (left) and the partition it induces (right): five regions $R_1, \dots, R_5$, each predicting the average response c_l of the training points it contains. Region fills and points share the same color scale, so a leaf's color is the blend of its points'.

In general, writing $R_1, \dots, R_L$ for the leaf rectangles and $|R_l|$ for the number of training points in leaf l, a regression tree computes

$$f(\mathbf{x}) = \sum_{l=1}^{L} c_l \, \mathbf{1}\{\mathbf{x} \in R_l\}, \qquad c_l = \frac{1}{|R_l|} \sum_{i:\, \mathbf{x}_i \in R_l} y_i, \tag{1}$$

a *piecewise-constant* function: constant on each rectangle, jumping at the thresholds. (A classification tree has the same form with c_l replaced by the leaf's majority class or its vector of class proportions.) Figure 3 shows what this means for a one-dimensional regression: the tree approximates a smooth curve by a *staircase*, coarse at depth 2, finer at depth 5. One consequence follows directly from the functional form: a tree has *no smoothness*: it cannot represent a slope, only approximate it by steps, so flexibility here means "any staircase", not "any smooth function". Two further consequences of the same form (indifference to feature scaling, and the inability

to extrapolate) are developed with the practical properties of section 2.3. How the questions themselves are chosen from data is the next section's business.

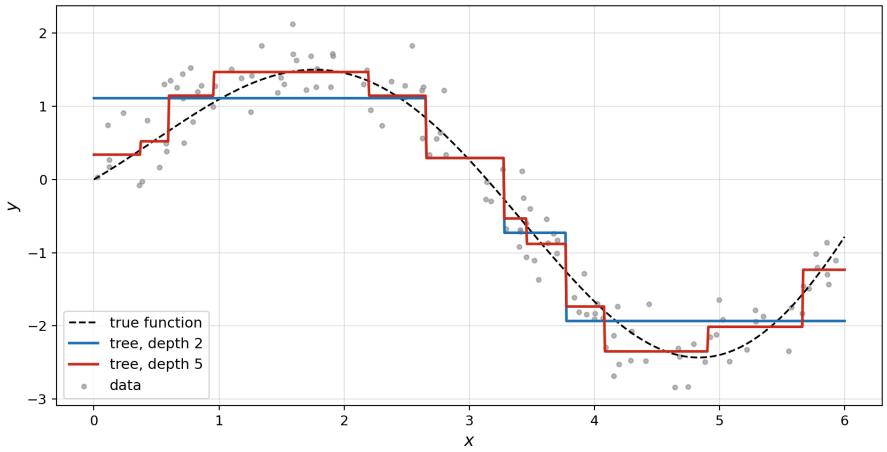


Figure 3: Regression trees are staircases: depth 2 gives four steps, depth 5 tracks the curve, and starts tracking the noise. Note the flat extrapolation outside the data range.

2 Growing a tree: the CART algorithm

Reading a tree is easy; the hard part is choosing the questions from data. This section states the learning problem, explains why it cannot be solved exactly, and develops the criterion actually optimized at each node, first for regression, where chapter 5's squared loss makes everything explicit, then for classification.

2.1 The regression criterion: variance

What should a good regression tree with J leaves achieve? Each candidate partition $R_1, \dots, R_J$ predicts, in region R_j, the average $\bar{y}_{R_j}$ of the training responses that fall in it (equation (1)). Its training error is the residual sum of squares, and the tree we would ideally want is the partition that minimizes it:

$$\min_{R_1,\dots,R_J} \sum_{j=1}^{J} \sum_{i:\,\mathbf{x}_i \in R_j} \left(y_i - \bar{y}_{R_j}\right)^2. \tag{2}$$

Solving (2) exactly is out of reach. The unknown is not a coefficient vector but the partition itself: every way of carving the feature space into J rectangles by nested axis-aligned cuts is a candidate, and their number grows combinatorially with the sample size, the dimension and J. There is also no gradient to descend: changing a threshold slightly either changes nothing or makes points jump between regions. CART (*classification and regression trees* [1]) therefore goes *greedy*: solve the problem for a *single* split, keep that split, and recurse on the two halves.

One split is tractable. A candidate split of a node is a pair (j, t), a feature and a threshold, sending the node's points with $x_{ij} \le t$ into a left child $L(j,t)$ and the rest into a right child $R(j,t)$. Applied to two regions, the objective of (2) becomes: choose the split whose two children are most *homogeneous*, each around its own average,

$$\min_{j,\,t} \sum_{i \in L(j,t)} \left(y_i - \bar{y}_L\right)^2 + \sum_{i \in R(j,t)} \left(y_i - \bar{y}_R\right)^2, \tag{3}$$

where $\bar{y}_L$ and $\bar{y}_R$ are the averages of the responses landing in each child. Problem (3) is solved by an exhaustive scan, organized so that it stays cheap. Only thresholds that separate two observed values of feature j produce different children, so the midpoints between consecutive sorted values

suffice: at most $n-1$ candidates per feature. For one feature, sort its values and sweep the threshold from left to right; each step moves a single point from the right child to the left, so running sums of the y_i and y_i^2 update both averages and both sums of squares in constant time. The whole scan over d features costs about d sorts of n values. The best (j, t) is kept, the node is split, and the same scan runs again inside each child.

It is customary to report a split not by its raw score (3) but by how much it *improves* on the unsplit node. Dividing the sums of squares by the node's size turns them into empirical variances ($\mathrm{Var}(S) = \frac{1}{|S|}\sum_{i\in S}(y_i - \bar{y}_S)^2$ for a node holding the points S), and minimizing (3) is equivalent to maximizing the *variance decrease*

$$\Delta(j,t) = \mathrm{Var}(\text{node}) - \big[w_L\,\mathrm{Var}(\text{left}) + w_R\,\mathrm{Var}(\text{right})\big], \tag{4}$$

where w_L, w_R are the fractions of the node's points sent to each child. The chapter-4 reading holds throughout: the average is the constant that minimizes a leaf's squared error, the minimal error of a node *is* its variance, and Δ is exactly the reduction in training squared error the split achieves. The gain is always nonnegative: it equals the between-children spread of the two leaf means, zero only when the split leaves both means equal.

Figure 4 shows the scan at work on one node of 26 points. Three candidate thresholds on the same feature earn very different gains: $t = 2$ separates a low group but leaves the mixed middle together ($\Delta = 0.99$); $t = 5$ falls exactly between the two levels and shrinks the node variance from 2.72 to a weighted 0.72 ($\Delta = 1.99$); $t = 8$ cuts through the high plateau and gains almost nothing ($\Delta = 0.11$). The sweep evaluates every candidate and keeps $t = 5$.

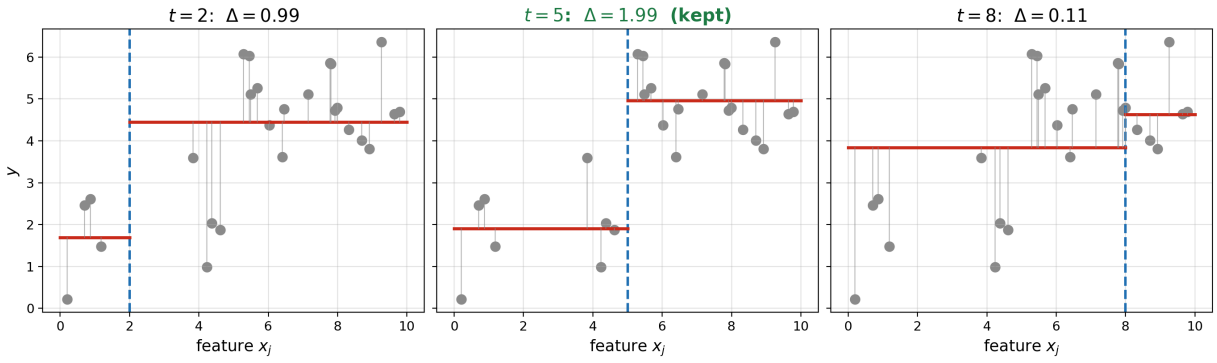


Figure 4: One node, three candidate splits under the regression criterion. The red segments are the two leaf means, the thin lines the residuals they leave; the middle threshold earns the largest variance decrease and is kept.

2.2 The classification criterion: impurity

For a discrete target there is no average to fit and no residual to square, so the score of (3) needs a replacement. The most natural candidate is the training *misclassification error*: a node predicts its majority class, so a binary node with class-1 proportion p errs on a fraction $\min(p, 1-p)$ of its points, and a split could be scored by how much it lowers the children's weighted error. This obvious choice is discarded in practice. The function $\min(p, 1-p)$ is piecewise-linear (dashed in figure 5), and along a linear segment the children's weighted error adds up to exactly the parent's: any split whose two children keep their proportions on the same side of $1/2$ (both still majority class 0, say) scores *zero improvement*, even when one child came out far purer than the other. A greedy search fed by this score stalls: it cannot tell a useless split from a genuinely useful one that does not yet flip a majority. This is why the misclassification rate is kept for *evaluating* a finished tree, never for growing it.

What the search needs instead is a score that rewards progress toward *purity* itself. Purity is quantified by an *impurity function* of the class proportions in the node; the two standard choices

(Gini from the CART tradition [1], entropy from the ID3 family of tree learners [2]) for a binary node with class-1 proportion p:

$$\text{Gini:}\quad G(p) = 2p(1-p), \qquad \text{entropy:}\quad H(p) = -p\log_2 p - (1-p)\log_2(1-p).$$

The Gini index is the probability that two members of the node, drawn at random, belong to different classes: zero for a pure node, since two draws then always share the same class. The entropy is information theory's measure of surprise: the average number of bits needed to announce the class of a member; a pure node costs zero bits, a 50/50 node costs one full bit. Both vanish on pure nodes, peak at the even mixture, and are strictly *concave*. Figure 5 makes the numbers concrete on three nodes of 20 points: the 50/50 node sits at the top of both curves ($G = 0.50$, $H = 1.00$), the 15-vs-5 node has dropped to $G = 0.38$, $H = 0.81$, and the 19-vs-1 node is nearly pure ($G = 0.10$, $H = 0.29$).

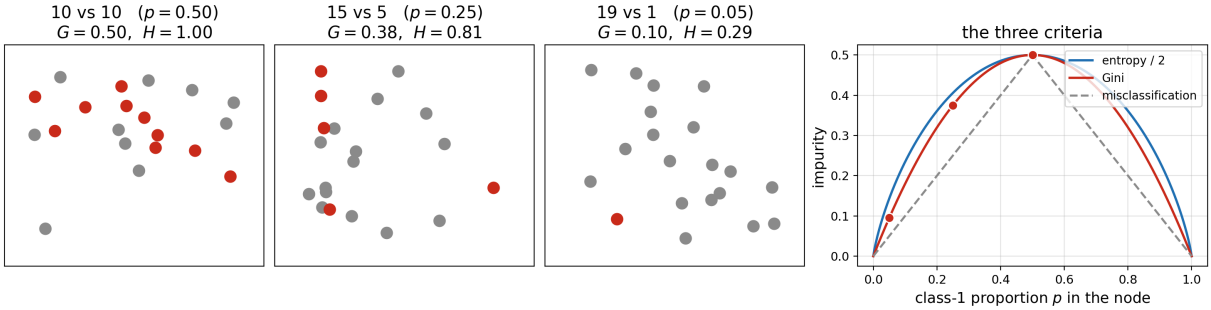


Figure 5: Purity, quantified. Three nodes of increasing purity with their computed Gini and entropy (left), and the three criteria as functions of the class-1 proportion (right; dots mark the three nodes). Gini and (rescaled) entropy are strictly concave; the misclassification rate is piecewise-linear, which is what disqualifies it as a growing criterion.

The quality of a split (j, t) has the same weighted-decrease shape as (4), with the impurity in the role of the variance: the *impurity decrease*

$$\Delta(j,t) = \text{imp}(\text{node}) - \big[w_L\,\text{imp}(\text{left}) + w_R\,\text{imp}(\text{right})\big]. \tag{5}$$

Strict concavity is exactly what the misclassification rate lacked: the children's weighted impurity now sits strictly below the chord joining their two positions on the curve, so $\Delta \geq 0$ for every split, with strict progress whenever the split changes the class mix, including the splits the misclassification rate scored at zero. Gini and entropy, nearly proportional in figure 5, almost always pick the same splits; Gini is the common default (no logarithms to evaluate). The greedy machinery is otherwise unchanged: same midpoint scan feature by feature, same recursion; only the score differs. Figure 6 repeats the three-threshold scan of figure 4 with the Gini criterion, on a node at $G = 0.49$: $t = 2$ splits off a small pure class-0 group ($\Delta = 0.12$), $t = 8$ isolates a pure class-1 tail on the right ($\Delta = 0.11$), and $t = 5$, sitting at the class boundary, wins with $\Delta = 0.30$.

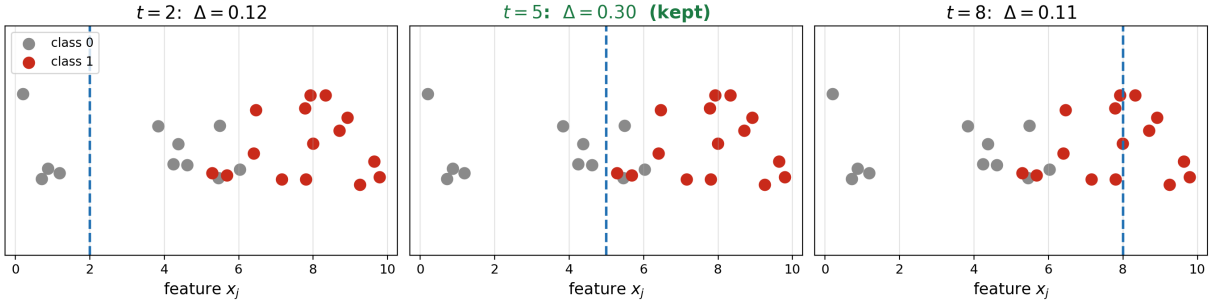


Figure 6: One node, three candidate splits under the Gini criterion. Each threshold improves purity; the one at the class boundary improves it most and is kept.

Two things remain visible once the recursion has run its course, and figure 7 displays both on a small finished tree. First, purity improves down the tree: the root's even mixture ($G = 0.50$) splits into two lopsided children ($G = 0.38$ each), and these split again into leaves that are pure or nearly so. Second, the leaves produce probabilities with no extra machinery: a leaf holding n_l training points of which n_1 are class 1 predicts the simple frequency $\hat{p} = n_1/n_l$. The leaf collecting 3 class-1 points out of 4 predicts 0.75, and the pure leaves predict 0 and 1. These frequencies are the class proportions promised in section 1.1, and they are the $\hat{p}$ that chapter 6's probability tooling (thresholds, calibration) consumes.

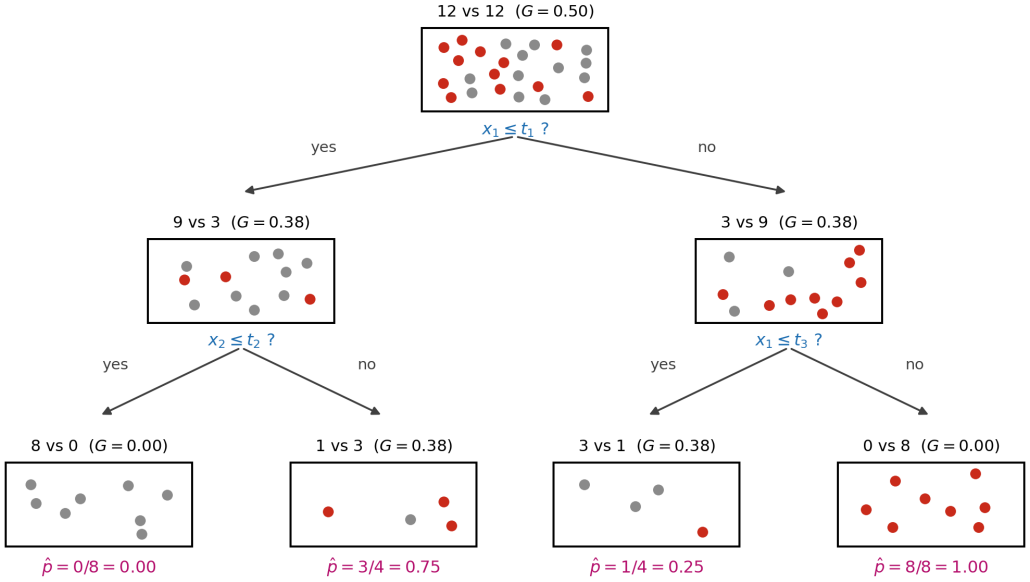


Figure 7: A small classification tree drawn with its data. Down the tree, the Gini index falls and the nodes grow purer; at the bottom, each leaf predicts the frequency of class 1 among its own points: pure leaves say 0 or 1, mixed leaves 0.25 or 0.75.

2.3 Scaling, missing values, categories, extrapolation

The growing mechanics settle four practical questions that chapter 6's pipeline had to handle with dedicated machinery.

No standardization. Every split compares one feature to a threshold, so only the *order* of each feature's values matters. If φ is any increasing transformation (a log, a z-score, a min–max rescaling), then $x_j \le t$ holds exactly when $\varphi(x_j) \le \varphi(t)$: the transformed feature offers the same candidate partitions with the same gains, and the grown tree is *identical*, threshold labels aside. The scaling debates of chapter 6 (standardize or min–max, before which penalty) do not arise: trees have no penalty comparing coefficients and no distance adding coordinates, the two mechanisms that made scaling matter.

Missing values, natively. Chapter 6 imputed the missing ages before anything else could run. A tree can instead treat missingness inside the split itself (figure 8): a row whose feature j is missing cannot answer "$x_j \le t$?", so during growth the algorithm tries routing all such rows left, then right, and keeps the direction with the lower loss. This *learned default direction* is stored with the split and followed again at prediction time. No imputation constant is estimated, and missingness is free to carry signal, as chapter 6's indicator column intended, but decided locally,

split by split. XGBoost's sparsity-aware splits and LightGBM's missing handling implement this mechanism.

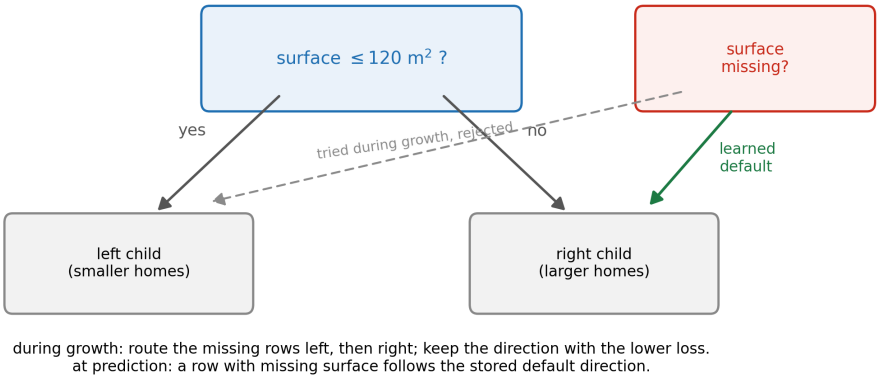


Figure 8: Native missing-value handling. Rows whose split feature is missing are routed left, then right, during growth; the lower-loss direction becomes the split's stored *default*, followed at prediction time. No imputation step is needed, and informative missingness is exploited automatically.

Categorical features. A district or a heating type has no order, hence nothing to threshold. Two routes coexist. The simplest is chapter 6's one-hot encoding: each category becomes a 0/1 indicator column, a split on that column ("$\mathbf{1}\{\text{district} = \text{Marina}\} \leq 0.5$?") is an ordinary threshold question meaning "is the listing in Marina or not", and growing proceeds with no new machinery. The alternative, offered by CART implementations with native categorical handling, lets a single split send a *subset* of categories left and the rest right: "district $\in$ {Marina, Center}?". The $2^{K-1} - 1$ category subsets need not all be scanned: for a binary target or a regression, sorting the K categories by their mean response and sweeping that ordering like a numerical feature finds the best subset split, with $K - 1$ candidates instead of exponentially many.

No extrapolation. The tree's flexibility inside the training range has a hard boundary at its edge. Every prediction is some leaf's constant, and past the largest training value the active leaf never changes: the prediction *freezes*. Figure 9 fits a regression tree and a linear model to the same data with a clean linear trend, then evaluates both beyond the observed range: the line continues the trend (here almost exactly, 20.5 predicted at $x = 16$ against a truth of 20.6), while the tree answers 13.3 (its last leaf's mean) for every x past the boundary, however far. A linear model can extrapolate wrongly, but it *moves*; a tree cannot.

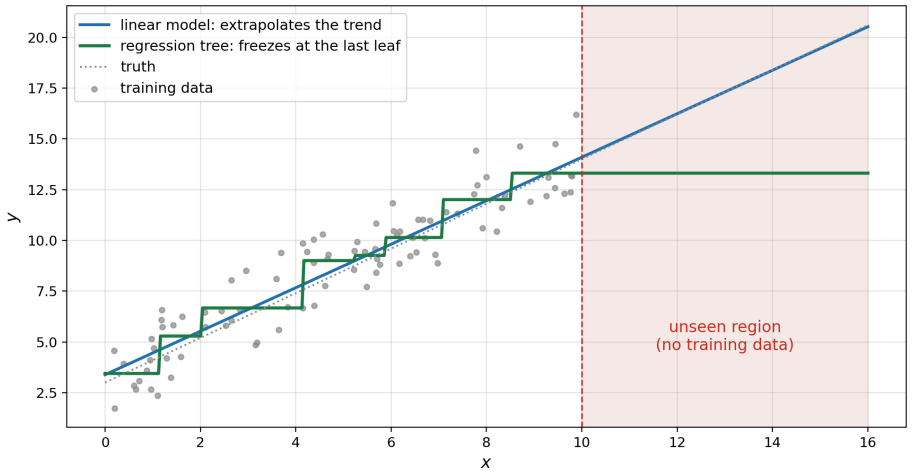


Figure 9: Extrapolation. On data with a linear trend, the fitted line continues into the unseen region and tracks the truth; the regression tree is frozen at its boundary leaf's constant. Inside the training range, the roles are less clear-cut: the staircase can track structure a line cannot.

Reality check

The no-extrapolation property is a deployment hazard, not a curiosity. A regression tree trained on chapter 6's housing file has never seen a 300 m^2 listing and will price every such property like the largest one it did see; a tree trained before a market-wide price rise will keep predicting the old level, however far prices drift. The monitoring discipline of chapter 6 (drift alarms, retraining) is not optional for deployed trees.

2.4 The CART algorithm in pseudocode

Everything in this section assembles into one short recursive procedure. Its input is the set S of training rows reaching the current node; the first call receives the full training set.

Algorithm GROW(S):

1. **Stop?** If a stopping rule fires, namely S is pure (all responses identical), $|S|$ is below a minimum node size, or a maximum depth is reached, then return a leaf carrying the average response of S (regression) or its class proportions (classification).
2. **Scan.** For every feature j and every midpoint threshold t between consecutive sorted values of feature j in S (a subset split for a categorical feature), score the candidate: the children's residual sum of squares (3) for regression, the impurity decrease (5) for classification; rows missing feature j follow the better of the two default directions.
3. **Split.** Keep the best pair (j^*, t^*) and partition S into $L = \{i \in S : x_{ij^*} \le t^*\}$ and $R = S \setminus L$.
4. **Recurse.** Attach the subtrees GROW(L) and GROW(R) to the node and return it.

The stopping rules of step 1 are not a footnote: they decide how far the recursion runs, hence how complex the tree gets. Choosing them well is the next section's subject.

3 Complexity, generalization, hyperparameters

Depth and the U-curve. A tree of depth q can carve up to 2^q rectangles; deep enough, it isolates every training point in a pure leaf and memorizes the sample. Figure 10 replays chapter 5's U-curve with depth as the complexity parameter: on the two-moons data, depth 2–3 achieves

10% test error, depth 12 drives the training error to its floor while the test error climbs to 13%, the boundary sprouting rectangular islands around individual points. This is chapter 5's Ilyas rule, grown by an algorithm. Nothing about tree overfitting is new: it is the same U-curve with a new complexity parameter, the third in the course's series (polynomial degree, regularization strength, and now tree depth).

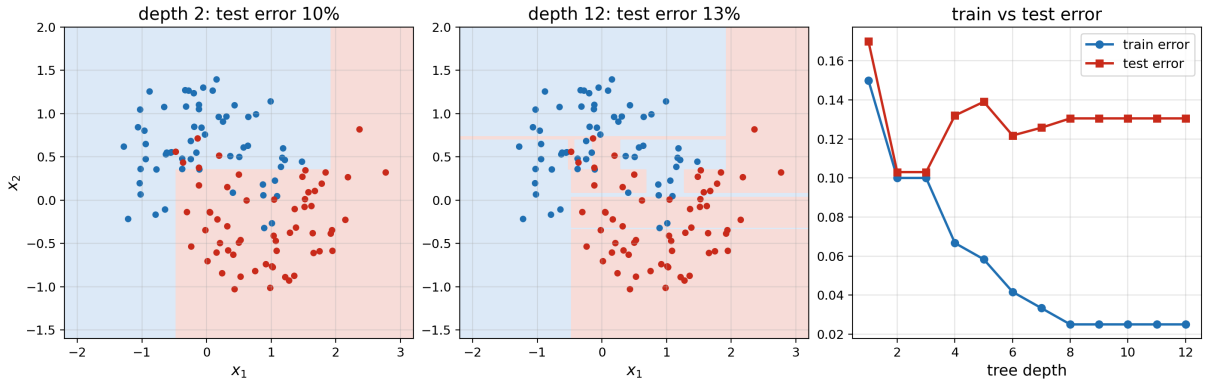


Figure 10: Depth as complexity on 120 training points: honest blocks at depth 2, memorized noise at depth 12, and the U-curve between them.

The tree's dials. Depth is only the most visible of the hyperparameters that bound a tree's flexibility, and the standard implementations expose two families.

- *Maximum depth* (or maximum number of leaves) caps the partition wholesale: at most 2^q rectangles, at most q features interacting along any root-to-leaf path.
- *Minimum samples per leaf* refuses any split that would leave fewer than k training points in a child. Every leaf value is then an average over at least k responses, which floors the variance of the leaf estimates and makes single-point "memorization leaves" impossible: a complexity cap by local sample size rather than by shape, bounding the number of leaves by n/k wherever the depth cap does not bite.

Both are hyperparameters in the precise sense of chapter 5 (settings of the procedure, not coefficients of the fit), and they are set with exactly chapter 5's machinery: candidate values, validation or cross-validated error, minimum; the whole pipeline of chapter 6 refitted per fold.

Instability. The greedy construction has a second, subtler cost. Each split is a hard argmax over data-dependent scores; perturb a few observations and the winning first split can change, taking the entire subtree with it. Trees are high-variance estimators in the precise sense of chapter 5's bias–variance trade-off, and, unusually, the variance shows up as *structural* volatility: the printed rules themselves change from sample to sample, which undermines the interpretability that motivated the tree in the first place. Variance, and the cost of reducing it, drive the rest of the chapter.

4 Bagging and random forests

4.1 Bagging: averaging the bootstrap refits

Averaging reduces variance: chapter 1's first formula, $\mathbb{V}(\bar{X}) = \sigma^2/n$, says that the average of n independent noisy quantities is less noisy than any one of them. This suggests a remedy for the instability of trees. If B deep trees could be fitted on B *independent* samples, their average would keep the trees' low bias while dividing the variance by B. Only one sample is available.

This is the situation chapter 4's bootstrap was built for, and its device applies unchanged: draw n rows with replacement from the training set to form a bootstrap replica, and repeat B times. *Bagging* (*bootstrap aggregating*) [3] fits one deep tree on each replica and averages the B predicted probabilities; for classification one can also take a majority vote of the predicted classes. Chapter 4 used the refits on the replicas to estimate the spread of an estimator; bagging uses the same refits as B predictors and averages them.

Figure 11 shows the payoff as trees accumulate: a lone bootstrap tree is as jagged as the original (10.8% test error, against 13.1% for the deep tree on the full sample); the boundary steadies through $B = 10$ and $B = 100$, and by $B = 200$ the test error has fallen to 8.8%, below the best single tree of *any* depth, because averaging removes variance without incurring the bias of shallowness. Note also what the progression does *not* show: there is no U-curve in B. From $B = 100$ onward the error only edges down and flattens. Increasing B never causes overfitting; it only adds computation for smaller and smaller gains. Unlike the depth, B is therefore not a complexity hyperparameter to be tuned against overfitting: in practice one grows trees until the error curve has flattened, or until the computation budget is spent.

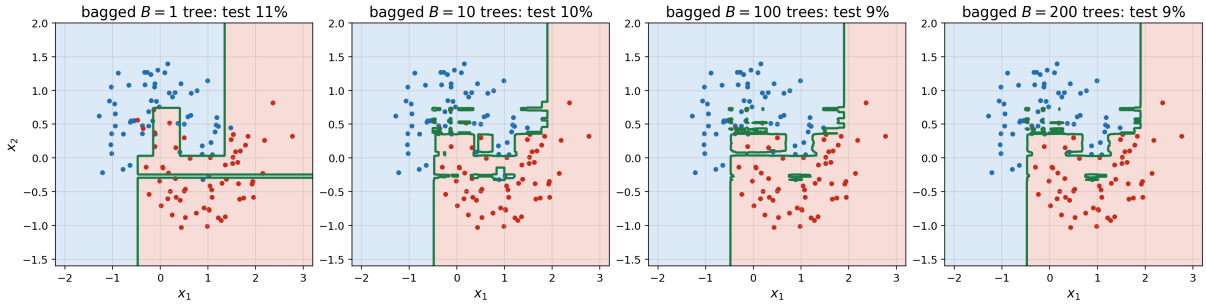


Figure 11: Bagging deep trees on the same 120 training points: the jagged boundary of a single bootstrap tree steadies as trees accumulate ($B = 10$, 100, 200), and the test error falls from 11% to 9% and flattens, with no overfitting in B.

The out-of-bag error. The bootstrap replicas also provide a built-in validation estimate. Each replica omits about 36.8% of the training points; equivalently, each training point is left out of about a third of the replicas (here, 73 of the $B = 200$). For a training point i, consider only the trees whose replica omitted it. These trees never saw point i, so their aggregated prediction at $\mathbf{x}_i$ is honest in chapter 5's sense: no row is graded by a fit it helped build. The average error of these predictions over the training set is the *out-of-bag* (OOB) error. It is a validation estimate obtained during training, with no held-out rows and no extra fitting; it plays the role cross-validation played in chapter 5, at no cost. On the moons run the OOB error is 12.5%, against the 8.8% measured on the large test set. Two effects explain the gap. The OOB error is computed on the 120 training points only, and an error rate estimated from 120 rows fluctuates by about three points in either direction; and each OOB prediction aggregates only about a third of the trees, which makes the estimate slightly pessimistic (the same direction as cross-validation's $n(1 - 1/K)$ pessimism in chapter 5). On datasets of realistic size both effects shrink. The OOB error is the standard way to monitor a bagged ensemble, and the practical way to set B: grow trees until it flattens.

4.2 Random forests: decorrelating the trees

The variance computation that motivated bagging assumed *independent* trees, and bootstrap trees are not independent: fitted on overlapping data, they often agree on their first splits, so their predictions are *correlated*. For identically distributed predictors with variance σ^2 and pairwise

correlation ρ,

$$\mathbb{V}\left(\frac{1}{B}\sum_{b=1}^{B} T_b\right) = \rho\,\sigma^2 + \frac{1-\rho}{B}\,\sigma^2 \tag{6}$$

where the second term vanishes as B grows; the first, the *correlation floor*, does not. Past a few hundred trees, adding more changes almost nothing; the only way to reduce the variance further is to lower ρ.

Lowering ρ means making the trees more different from one another, and the bootstrap alone does not go far enough: grown greedily on replicas that share most of their rows, the trees keep finding the same strong first splits, so their errors remain correlated. *Random forests* [4] add a second randomization, inside the growing itself: *feature subsampling*. At *every split*, only a random subset of $m \leq d$ features is eligible to be scanned (the `max_features` parameter of the standard implementations, typically $m \approx \sqrt{d}$ for classification). The strongest feature is then unavailable at many splits, different trees are forced to use different features, and their errors decorrelate. The forest predictor is the average of B such randomized bootstrap trees,

$$\hat{F}_B(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} T_b(\mathbf{x}), \tag{7}$$

where T_b is a deep tree grown on the b-th bootstrap replica under per-split feature menus. For classification, each $T_b(\mathbf{x})$ is the tree's vector of leaf class proportions; the average is thresholded, or a majority vote taken. Each individual tree gets slightly worse, since its best split is often not available; the average gets better: variance traded against bias one more time, now *inside* the ensemble. Everything else carries over from bagging unchanged, the out-of-bag error included: it remains the standard way to monitor a forest.

5 Gradient Boosting Machines

5.1 Sequential construction

Boosting is the second way of combining trees, and it differs from the random forest on every design choice. The forest grows B *deep* trees *independently* (each on its own bootstrap replica, none aware of the others, so all of them can be built in parallel) and averages their predictions to cancel variance. Boosting grows its trees *sequentially*: each tree is deliberately *small* (a *weak learner*, too simple to fit the data on its own) and is fitted to correct the errors left by the trees built before it. The prediction is the sum of these corrections, not an average of full fits.

By hand. The clearest way to see the mechanism is regression with squared loss, done by hand on figure 12. Fit a first small tree (depth 2) to the data: a coarse staircase F_1 (top-left panel). Its errors are the residuals $r_i = y_i - F_1(\mathbf{x}_i)$, the vertical gaps between the points and the staircase, and they are not pure noise: they still contain the part of the curve that F_1 missed. Correcting F_1 therefore means predicting its residuals, and that is just another regression problem. Fit a second small tree to r and add it to the first, damped by a *learning rate* η; the top-right panel shows the updated fit $F_2 = F_1 + \eta\,\text{tree}_2$, already closer to the data. Repeating the same step (compute the current residuals, fit a small tree to them, add it) gives the general update:

$$F_m = F_{m-1} + \eta\,\text{tree}_m, \qquad \text{tree}_m \text{ fitted to } \; y - F_{m-1}. \tag{8}$$

After 5 rounds the staircase tracks the curve; after 50 it tracks it finely (bottom panels). No tree in the sequence fits the data on its own; the sum of M small damped corrections does. And because every tree is fitted to its predecessors' residuals, the trees cannot be built in parallel: the order of construction is part of the model.

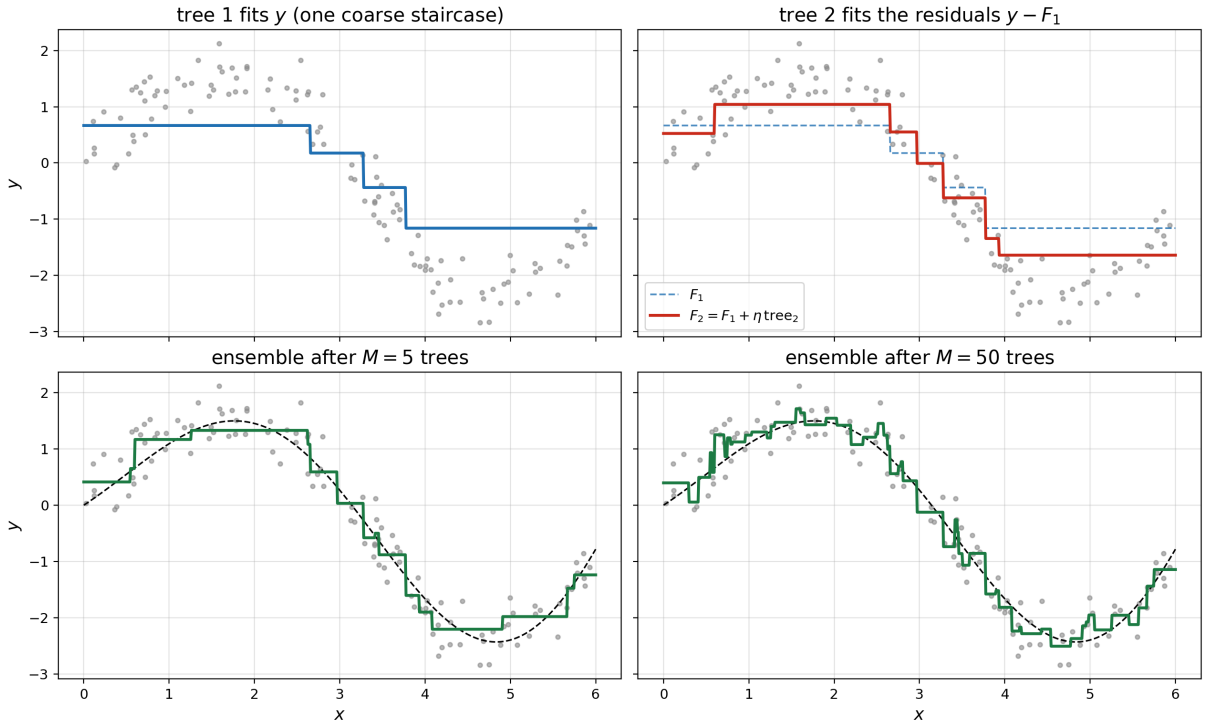


Figure 12: Boosting by hand with depth-2 trees, every panel on the same data and scale: the first staircase F_1; the residual tree's damped update $F_2 = F_1 + \eta\,\text{tree}_2$; the ensemble after 5 and 50 rounds.

5.2 Gradient boosting

From residuals to gradients. Fitting the residuals looks specific to the squared loss; it is in fact an instance of a general principle. For the squared loss $\ell = \frac{1}{2}(y - F)^2$, the residual is exactly the negative derivative of the loss with respect to the prediction: $-\partial\ell/\partial F = y - F$. Each round of boosting therefore takes a step in the direction that decreases the loss fastest: the gradient logic chapter 5 used to derive OLS, now applied in *function space*, with a small tree as the step. This generalizes immediately: for any differentiable loss, fit each tree to the *pseudo-residuals* $-\partial\ell/\partial F(\mathbf{x}_i)$. For chapter 5's log loss these are $y_i - p_i$, label minus predicted probability, which is how *gradient boosting* does classification: trees fitted to $y - p$, summed into a score, squashed by a sigmoid or softmax.

Overfitting in M, and early stopping. Adding trees to a random forest was harmless: B is a computation budget, not a complexity parameter, because averaging more replicas of the same recipe can only stabilize the estimate (section 4). Boosting has no such safety. Every round is fitted to whatever the current ensemble still gets wrong, so the training loss decreases by construction, round after round, and once the learnable structure is exhausted, the rounds continue, on noise. Figure 13 shows both losses along a single boosting run ($\eta = 0.1$, depth-2 trees): the training MSE falls throughout, from 0.083 at round 51 to 0.049 at round 400, while the validation MSE bottoms out at $M = 51$ (0.132) and climbs back to 0.149. The remedy is read directly off the picture. Hold out a validation set, evaluate it after every round (each round adds one tree, so the entire curve costs a single training run) and stop once the validation loss has stopped improving; implementations wait a fixed number of rounds without improvement (a *patience* window) before declaring the minimum and keep the ensemble as it was at that round. This is *early stopping*, and it is how the length of a boosting sequence is chosen in practice.

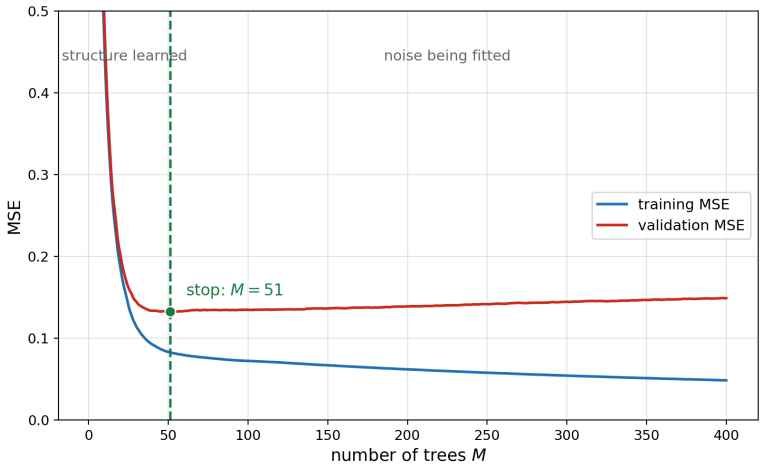


Figure 13: Early stopping. Along one boosting run, the training MSE decreases at every round by construction; the validation MSE turns back up once the rounds start fitting noise. Training is stopped at the validation minimum, here $M = 51$.

The three knobs. Gradient boosting has three coupled hyperparameters.

- *Learning rate* η. Damping each tree's contribution forces the ensemble to approach the data slowly, spreading a given correction over many trees. Figure 14 shows the effect: $\eta = 1$ reaches its best test error fast (0.169 at $M = 27$ trees) then *overfits visibly* as further trees fit noise (0.223 at $M = 400$); $\eta = 0.05$ needs ~ 100 trees but attains a better minimum (0.148) and barely degrades afterwards. A small η costs more trees, so the practical default is a small learning rate with many trees.
- *Number of trees* M. Set by the early stopping of the previous paragraph, not by a grid: cap M generously and let the validation curve pick the stopping round.
- *Tree depth.* Each tree's depth bounds the *interaction order* the ensemble can express: stumps (depth 1) yield additive models in single features; depth q allows effects involving up to q features jointly. Typical values 3–8: deep enough for interactions, shallow enough to stay weak.

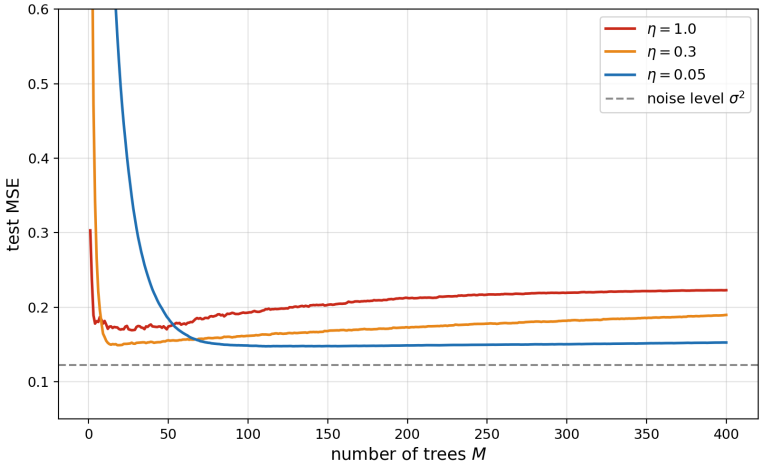


Figure 14: Test error against the number of trees for three learning rates: large η overfits early and hard; small η is slower, better, and far more forgiving, with M set by early stopping.

Gradient boosting as described dates from the late 1990s [5]. It reached wide use through a pair of industrial implementations of the same principle, each with its own signature ideas: XGBoost

refined the objective and the systems engineering; LightGBM redesigned the computation for speed.

5.3 XGBoost

XGBoost ("extreme gradient boosting", 2016) [6] refines the boosting loop in two directions.

Curvature and built-in regularization. Where classical boosting uses only the loss gradient, XGBoost uses its curvature too. Writing g_i and h_i for the first and second derivatives of the loss at the current prediction (for the log loss, $g_i = p_i - y_i$ and $h_i = p_i(1 - p_i)$), the loss of adding a tree f is approximated to second order and the tree is chosen to minimize

$$\sum_{i=1}^{n} \left[g_i f(\mathbf{x}_i) + \tfrac{1}{2} h_i f(\mathbf{x}_i)^2 \right] + \gamma T + \tfrac{\lambda}{2} \sum_{j=1}^{T} w_j^2, \tag{9}$$

where T is the number of leaves and w_j the leaf values. Note the *built-in regularization*: a penalty γ per leaf and a ridge penalty on the leaf values, chapter 5's penalization now applied inside every tree. For a *fixed* partition the optimum is closed form: leaf j, collecting $G_j = \sum_{i \in j} g_i$ and $H_j = \sum_{i \in j} h_i$, gets

$$w_j^* = -\frac{G_j}{H_j + \lambda}, \tag{10}$$

and the resulting gain formula scores candidate splits, so the greedy growth of section 2 proceeds with a loss-aware, regularized criterion instead of raw impurity. (Sanity check: for the squared loss $h_i = 1$, and at $\lambda = 0$ the leaf value w_j^* is the mean residual; the residual-fitting update of section 5 is recovered.)

The engineering. The rest of XGBoost's advantage is systems work aimed at large tabular data: *row and column subsampling* per tree (bagging's and the forests' ideas imported into boosting, decorrelating trees and regularizing); *approximate split finding*, which scans quantiles of each feature instead of every threshold (a weighted sketch keeps this principled); *sparsity-aware splits*, the learned default directions of section 2.3; and *within-tree parallelism*. The scan over candidate features at a node is embarrassingly parallel, so split search uses all cores (the trees themselves remain sequential: boosting cannot parallelize over M), with columnar, cache-friendly data layout making the scans fast.

Figure 15 makes the algorithm concrete on the moons data of figure 11, with panels at the same counts (boosting rounds now instead of bagged trees). One round is one depth-3 tree: a coarse corner. At $M = 10$ the boundary has barely moved: with $\eta = 0.1$, ten damped corrections still resemble the first tree. By $M = 100$ the interleaving of the moons is carved out and the test error reaches 8%, below bagging's 8.8%. And at $M = 200$ the error ticks back up to 8.5%: the sequence has started fitting noise. Set against figure 11's flat progression in B, this uptick is what early stopping exists to catch.

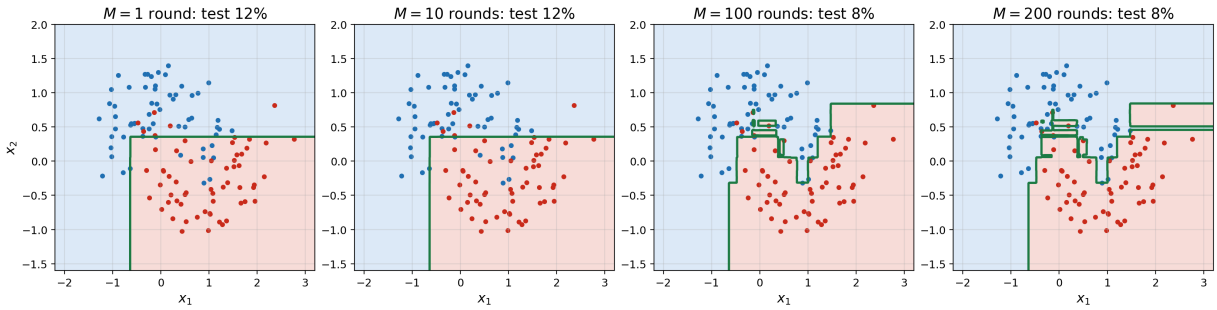


Figure 15: XGBoost on the moons data of figure 11 ($\eta = 0.1$, depth-3 trees). The boundary barely moves over the first damped rounds, carves the moons by $M = 100$ (8%, below bagging's 8.8%), and starts fitting noise by $M = 200$ (8.5%): boosting overfits in M where bagging stayed flat in B.

5.4 LightGBM

Microsoft's *LightGBM* (2017) [7] keeps XGBoost's framework and reduces its computational cost, with two signature choices.

Histograms. Consider what the split scan of section 2 costs at boosting's scale. Growing one node means sweeping the sorted values of every feature, an effort proportional to the number of observations times the number of features, affordable once, but boosting repeats it at every node of every tree, and a tuned booster on a large file easily means hundreds of trees with dozens of nodes each, over millions of rows. The exact threshold search becomes the dominant cost of training. LightGBM's answer is to give up exactness: each feature is *pre-binned* into a small number of histogram buckets (255 by default) once, before training. Split search at a node then costs one pass over the bins, not over the observations; memory drops (one byte per value), and a sibling's histogram is obtained for free by subtracting the child's from the parent's. The loss of resolution (thresholds can only sit at bin edges) turns out to be harmless in practice and even mildly regularizing. The idea proved so effective that XGBoost later adopted a histogram mode, and scikit-learn's `HistGradientBoosting` estimators are built on the same principle. Two further LightGBM economies push in the same direction: gradient-based sampling keeps all large-gradient rows (the poorly-fitted ones) and subsamples the rest, and mutually exclusive sparse features are bundled into single columns.

Leaf-wise growth. The deeper difference is the *order* in which the tree grows (figure 16). Classical implementations grow *level-wise*: split every node of the current depth that can still be split, then descend. The result is capped by `max_depth` but not necessarily symmetric: a node that is already pure, or too small for the minimum leaf size, stays a leaf while its siblings keep splitting (two such nodes in the figure). LightGBM grows *leaf-wise* (best-first): at every step, split the *one leaf with the largest gain*, wherever it sits. For the same number of splits, the tree allocates them where the loss reduction is largest, producing deep, asymmetric branches in the hard regions of the space and leaving easy regions coarse: more loss reduction per split, and the main reason LightGBM typically trains faster to equal accuracy.

Figure 17 runs LightGBM on the same moons, at the same round counts as figure 15, and the signature difference is visible from the very first panel: a single 8-leaf leaf-wise tree already cuts a notch inside the gap between the moons, spending its leaves where the gain is, where XGBoost's level-wise first tree drew a plain corner. The later panels carve jagged, asymmetric pockets in the hard region and leave the easy regions coarse. On this small file the two boosters land at similar test errors (10% against 8%–8.5%); what the figures expose is a difference of style (where each algorithm spends its splits) more than of accuracy, which matches the practical verdict of the conclusion.

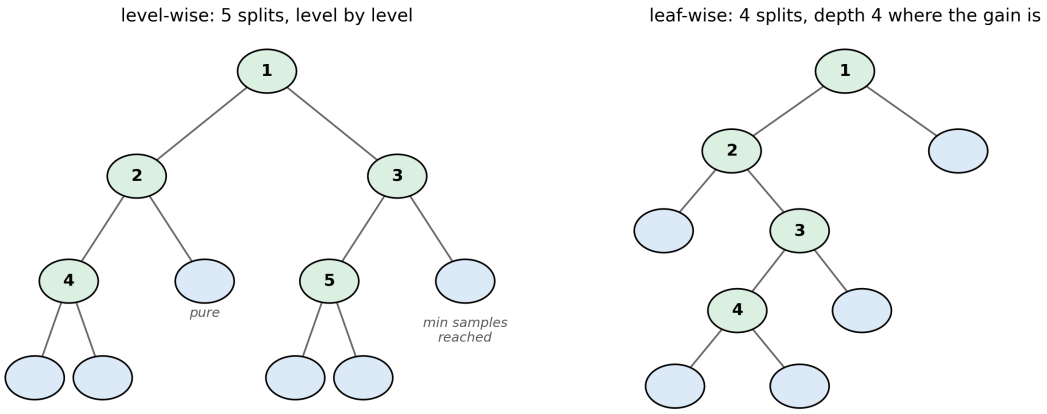


Figure 16: Growth orders (numbers = expansion order, green = split nodes, blue = leaves). Level-wise finishes a level before descending, without being symmetric: two nodes stopped early (one pure, one at the minimum leaf size). Leaf-wise follows the gain, producing deep asymmetric branches.

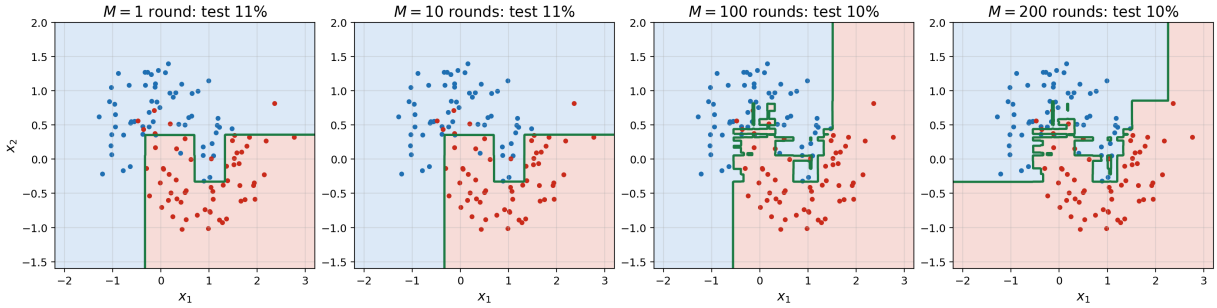


Figure 17: LightGBM on the same moons and round counts as figure 15 ($\eta = 0.1$, 8 leaves per tree). The first leaf-wise tree already carves inside the gap (its leaves go where the gain is) and later rounds cut deep, asymmetric pockets: a different style of boundary for a similar final error.

Overfitting in both families. Leaf-wise growth also overfits more easily: on small or noisy datasets, repeatedly splitting the highest-gain leaf produces deep, narrow leaves that fit individual points. LightGBM is therefore controlled by `num_leaves` (the complexity dial that actually binds; $2^{\text{max_depth}}$ is only an upper bound), a minimum leaf size, and optionally a depth cap. The discipline common to both families is the one this course has repeated since chapter 5: hold data out, monitor the validation loss, and *early-stop* the number of rounds; shrink η; subsample rows and columns.

6 Model comparison

6.1 Real data: the Boston housing benchmark

The chapter's models are now tested on real data, and the classic benchmark of the field is itself a housing problem: the *Boston housing* dataset [8]. It covers 506 census tracts of the Boston area (1978), the target being each tract's median home value (in thousands of dollars) and the features twelve tract descriptors: crime rate, residential zoning, industry share, a Charles-river flag, air pollution (NOx), mean rooms per dwelling, age of the housing stock, distance to employment centers, highway access, property tax, pupil–teacher ratio, and the share of lower-status population. Two chapter-6 reflexes apply before any model. The recorded target is *capped* at 50 k$ (sixteen tracts sit exactly at the cap, censoring in the wild), and one column of the

original file (an engineered function of the tract's racial composition) is ethically indefensible as a predictor; we exclude it.

The protocol is the standard one: 152 tracts held out as the test set; on the remaining 354, ridge and lasso tune λ by five-fold cross-validation, while the single tree and the boosters use a 100-tract validation split for depth (CART) and early stopping (XGBoost and LightGBM, depth-3 and 8-leaf trees, learning rate 0.05). The random forest runs at default settings ($B = 500$ deep trees, $m = d/3$ features per split) and needs no validation data at all: it is fitted on the full 354 tracts, with the out-of-bag error of section 4.1 as its only internal check (3.69 k$ here). Figure 18 shows the outcome, and this time the ensembles win. Ridge and lasso tie at a test RMSE near 4.6 k$ (the lasso keeps every column). A *single* validation-pruned tree already beats them at 3.65. The boosters improve further: LightGBM 3.19, XGBoost 3.08. And the random forest finishes first, at 2.66 k$, over forty percent below the linear models, with no analyst engineering a single feature and no hyperparameter tuned. Part of the forest's edge on this small file is exactly its low-maintenance design: it spends no tracts on a validation split, where the boosters give up 100 of theirs to early stopping.

The left panel adds the training errors, and it makes chapter 5's warning quantitative. Every tree model sits well below its own test bar: the forest trains at 2.04 k$ but tests at 2.66, XGBoost at 2.27 against 3.08; each flexible model absorbs part of the training tracts' noise, and its training figure overstates it accordingly. The linear models' two bars nearly coincide (4.85 against 4.62; this particular test draw even comes out slightly easier than the training tracts), because a family this rigid has little capacity to fit noise in the first place. The gap between the two panels grows with the model's flexibility; the ranking that counts is the test panel's.

The rightmost panel shows where the ensembles' gain lives: the ridge predictions bend away from the diagonal at both ends (overpricing the cheapest tracts, underpricing the most expensive, because a linear fit cannot follow the curved relationship), while the forest's predictions hug the diagonal all the way to the cap. Boston's structure is what tree models are built for: curved effects (each additional room is worth more in the upper range), thresholds, and interacting neighborhood factors, discovered automatically from twelve raw columns.

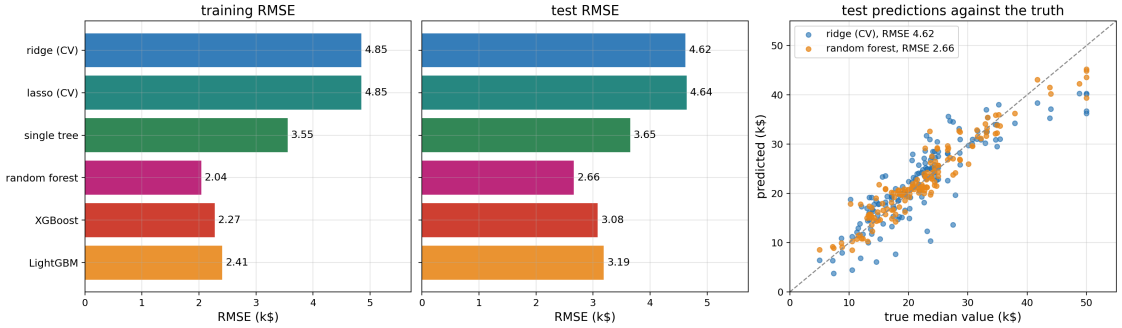


Figure 18: Six predictors of the median home value on the Boston tracts. Left and middle: training and test RMSE on a shared scale. The flexible models' training bars sit well below their test bars, the linear models' nearly coincide; on the test side, ridge and lasso tie near 4.6 k$, a single validation-pruned tree reaches 3.65, XGBoost 3.08, LightGBM 3.19, and the default-settings random forest 2.66. Right: test predictions against the truth. The ridge predictions bend away from the diagonal at both ends, the forest's track it up to the visible 50 k$ cap.

6.2 Interpretability

With linear models, effects are read off coefficients; a tree has no coefficients, and ensembles have thousands of splits. The standard summaries answer "which features matter?" in two different ways.

Impurity-based importance (MDI). Every split was chosen for its gain; credit that gain to the feature that made the split, sum over all splits (and, in an ensemble, over all trees), and normalize. The result, *mean decrease in impurity*, is free (it falls out of training) and widely reported, but it must be read with two caveats: it is computed on the *training* data, so a feature that helped memorize noise gets credit; and it is mechanically biased toward features offering many candidate thresholds (continuous or high-cardinality ones), which get more chances to win a greedy contest.

Permutation importance. A more honest measure [4] asks a counterfactual question: *how much does performance drop if this feature's information is destroyed?* On held-out data, shuffle the values of feature j across observations (breaking its link with the response while preserving its marginal distribution) and record the drop in test performance; repeat a few shuffles and average. Permutation importance is model-agnostic and measured in the model's own performance units. Figure 19 shows it for the winning random forest on the Boston test tracts: the lower-status share (+3.9 k$ of RMSE when destroyed) and the rooms per dwelling (+2.9) tower over everything (the two variables whose curved effects the ensemble exploited), followed far behind by air pollution, the pupil–teacher ratio, crime rate and distance to employment. The usual caveat about *correlated* features applies here too: pollution, industry share and distance to employment move together across Boston's neighborhoods, so their individual bars understate the shared neighborhood factor they carry, and near-zero bars (highway access, residential zoning) mean redundant-for-this-model, not irrelevant-in-the-world. Neither summary is a causal statement; both describe what *this model* uses, not what the world does.

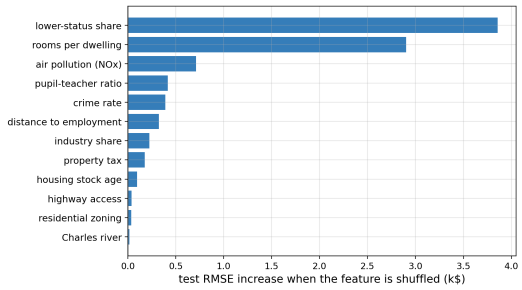


Figure 19: Permutation importance of the winning random forest on the Boston test tracts (RMSE increase per shuffled feature, averaged over eight shuffles). The lower-status share and the number of rooms dominate; correlated neighborhood descriptors split or hide their shared credit.

7 Conclusion

What is actually used in practice? On tabular data, gradient-boosted trees are the industry default: XGBoost and LightGBM power many winning entries of tabular competitions and production scoring systems, because they reach near-best accuracy while handling messy inputs natively (section 2.3) at a modest tuning cost. Random forests remain popular as the low-maintenance alternative: few hyperparameters that matter, no early stopping to manage, and the out-of-bag estimate for free. On the chapter's own small Boston benchmark, it was the untuned forest that finished first. Single trees are rarely deployed as predictors; they survive as communication devices, because anyone can read a printed tree. Linear models have not retired either. They are the standard baseline, and they remain the final model when coefficients, tests and predictable behavior matter more than the last points of accuracy, as in credit scoring under regulation.

How does an engineer compare all of these on a new problem? With a benchmark, built with the discipline of chapters 4 and 5. Freeze the splits first: a test set consulted once at the end, and a validation scheme for every choice. Fix the metric before any model runs. Then climb

a ladder of models, each evaluated under the identical protocol: a trivial baseline (predict the mean, or the majority class), then the linear pipeline of chapter 6, then an ensemble with default settings, and finally a tuned booster with early stopping and a small grid over learning rate, tree size, minimum leaf size and subsampling. The ladder is cheap and serves two purposes. It measures what each level of complexity is worth on this problem, instead of assuming it: a small gap between the linear rung and the tuned ensembles says the signal is mostly linear; a large gap, as on the Boston benchmark, says it is not. And it catches bugs: a trivial baseline with a suspiciously good score is usually a leak.

One practical choice remains: XGBoost or LightGBM? In accuracy they are usually equivalent, as they were on Boston (3.08 against 3.19). The real differences are elsewhere.

- *Speed and scale.* LightGBM is built to be fast on large files (histograms, leaf-wise growth, sampling). With millions of rows it is the usual first choice.
- *Small or noisy data.* XGBoost's level-wise trees are more conservative by default. LightGBM's leaf-wise growth digs deeper and overfits more easily, so `num_leaves` and the minimum leaf size need attention.
- *Features.* Both route missing values natively; LightGBM has the longer-standing native handling of categorical columns.

In practice, engineers run both with early stopping and keep the better validation score; the trial costs a few lines of code. When they tie, speed decides. Either way, the model is chosen on validation and evaluated once on the test set.

References

[1] L. Breiman, J. Friedman, R. Olshen and C. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[2] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[3] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

[6] T. Chen and C. Guestrin. XGBoost: a scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2016.

[7] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye and T.-Y. Liu. LightGBM: a highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30*, 2017.

[8] D. Harrison and D. L. Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, 5(1):81–102, 1978.