Quantitative Research Methods

Chapter 8: Tree-Based Methods

Prof. Dr. Christoph Weisser

HSBI

Summer Semester 2026

The course at a glance

	Lecture	Topic
	1	Ch. 1 + 2 — Introduction; what is statistical learning
	2	Ch. 2 — Accuracy, bias–variance, Bayes classifier, KNN
	3–4	Ch. 3 — Linear regression (estimation, inference, diagnostics)
	5–6	Ch. 4 — Classification (logistic, LDA/QDA, naive Bayes, ROC)
	7	Ch. 5 — Resampling (validation, CV, bootstrap)
	8	Ch. 6 — Model selection and regularisation (ridge, lasso)
	9	Ch. 7 — Moving beyond linearity (splines, GAMs)
▶	**10**	**Ch. 8 — Tree-based methods (trees, forests, boosting)**
	11	Ch. 10 — Deep learning (MLPs, CNNs, training)
	12	Ch. 13 — Multiple testing (FWER, FDR)

Twelve sessions of 180 minutes. ▶ marks today's chapter. Short exercises every ∼20 min; extended exercises every ∼45 min; each chapter has a companion lab notebook.

Contents

Optional and advanced material is collected in the **appendix** at the end of the deck.

Notation in this chapter

Symbol	Meaning
R_j	Rectangular region (leaf) of the partitioned predictor space
$\bar{y}_{R_j}$	Mean training response in region R_j (leaf prediction)
J, $\lvert T \rvert$	Number of leaves of a tree T (its complexity)
α	Cost-complexity pruning parameter: price paid per leaf
$\hat{p}_{mk}$	Fraction of node-m training points in class k
G	Gini index $\sum_k \hat{p}_{mk}(1-\hat{p}_{mk})$: node impurity
D	Cross-entropy $-\sum_k \hat{p}_{mk} \log \hat{p}_{mk}$: node impurity
B	Number of bootstrap samples / trees in an ensemble
m	Predictors tried per split in a random forest ($m = p$: bagging)
ρ	Pairwise correlation between the ensemble's trees
λ (or ν), d	Boosting learning rate (shrinkage) and tree depth

Sources and references

Primary textbook

James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). *An Introduction to Statistical Learning, with Applications in Python.* Springer.

About these slides

Each method is introduced with motivation, intuition, formal definition, and worked examples. Slide titles are descriptive.

Learning objectives for this chapter

By the end of this chapter you will be able to:

- **Build** regression and classification trees by recursive binary splitting; **apply** cost-complexity pruning.
- **Explain** why a single tree is high-variance and how *bagging* reduces that variance.
- **Distinguish** bagging from random forests; explain the benefit of feature subsampling at each split.
- **State** the boosting algorithm and its three tuning parameters.
- **Choose** between trees, random forests, and boosting for a given tabular problem.

Roadmap of this chapter

From single trees to industrial ensembles

1. Single trees: regression and classification, pruning.
2. Bagging: variance reduction by averaging.
3. Random forests: bagging plus feature subsampling.
4. Boosting: sequential additive trees on residuals.
5. BART: Bayesian sum of trees.

Outline

Why trees dominate tabular data

Tree-based methods are the dominant non-linear method on tabular data.

Five reasons

- Capture nonlinearity *and* interactions automatically.
- Interpretable as a sequence of yes/no questions.
- Handle mixed predictor types without dummy coding.
- Robust to outliers and monotone transformations.
- Ensembles (random forests, gradient boosting) routinely win Kaggle competitions on tabular data.

Single trees are weak: ensembles fix it

The single-tree pathology

A single tree is high-variance, blocky, and unstable. Small data changes produce very different trees.

Takeaway

The fix is to build many trees and aggregate. Bagging, random forests, gradient boosting, and BART all do this — in different ways.

Where this chapter is used in industry

Sector	Concrete application	Which decision it drives
Retail, supply chain	Demand forecast per article, store and week from price, promotion and calendar features	Replenishment orders, safety stock, promotion planning
Banking	Credit scoring, early-warning and collections prioritisation	Approve / decline, limit and risk-based pricing
Payments, e-commerce	Card-fraud and account-takeover scoring at transaction speed	Let through, challenge, or block — in milliseconds
Advertising, platforms	Click-through and conversion prediction, search and feed ranking	Which item to show, and what to bid
Insurance	Claim triage and fraud referral from claim, policy and text-derived features	Straight-through settlement or human adjuster
Manufacturing, telco	Predictive maintenance from sensor and service logs; churn propensity	Maintenance windows and retention offers

In industry — The common thread

Wide, mixed-type tabular data with missing values and interactions — the setting where boosted trees beat both linear models and neural networks, and where they are today the default choice.

Industry case in depth: retail demand forecasting

In industry — The model

Response: units sold per article × store × week (or day). **Predictors:** lagged and rolling sales, own and relative price, promotion and display flags, holidays and seasonality, article hierarchy and store attributes. Gradient boosting handles the mix natively — LightGBM-based solutions dominated the M5 forecasting competition.

What comes out — and who uses it

Not one number but a forecast *distribution* per article–store–week: the replenishment system orders against a service-level quantile, while the same forecasts feed the sales and operations planning round that finance and supply chain sign off. A store or category planner then releases or overrides each proposed order, with attributions (e.g. SHAP) explaining the unusual ones.

The honest caveat

Trees predict step functions, so they *cannot extrapolate*: a price point or a trend outside the training range gets clipped to the nearest observed level. Practitioners keep an explicit trend or price term, model in logs, and retrain on a cadence with drift monitoring.

Outline

Regression trees: idea

A regression tree is a piecewise-constant predictor over a partition of the predictor space.

The two-step view

1. Partition the predictor space into J rectangular regions $R_1, \ldots, R_J$.
2. Predict the mean response $\bar{y}_{R_j}$ on each region.

Goal

Minimise within-region variability: $\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \bar{y}_{R_j})^2$.

Reading the tree-fit objective

How to read this

Symbol	Meaning
J	Number of leaves (regions)
R_j	Rectangular region in $\mathbb{R}^p$
$\bar{y}_{R_j}$	Mean of y-values inside region j

Takeaway

Finding the globally optimal partition is computationally infeasible. In practice we use *recursive binary splitting*.

Recursive binary splitting: one step

The greedy step

At each node:

1. For every predictor X_j and every cutpoint s, compute the RSS (residual sum of squares) that would result from splitting on $X_j < s$.
2. Pick the (j, s) that minimises the resulting RSS.
3. Recurse on each child.

Greedy growth and stopping rules

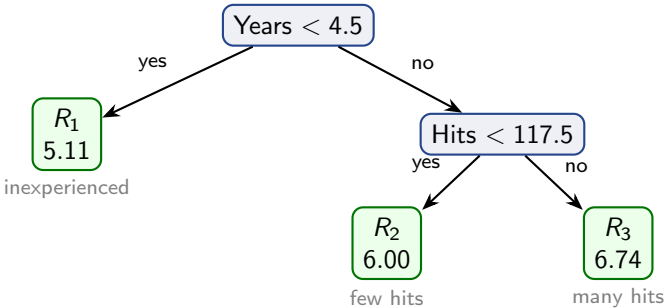
"Greedy" means each split optimises the *current* node only, never looking ahead — fast, but not the globally optimal tree (hence grow-then-prune). Stop on minimum leaf size, maximum depth, or minimum RSS gain.

Hitters: a baseball-salary tree

Years and Hits split the salary space into three regions; each leaf predicts the regional mean log-salary.

Source: Figure 8.1 — ISLP, James et al. (2023).

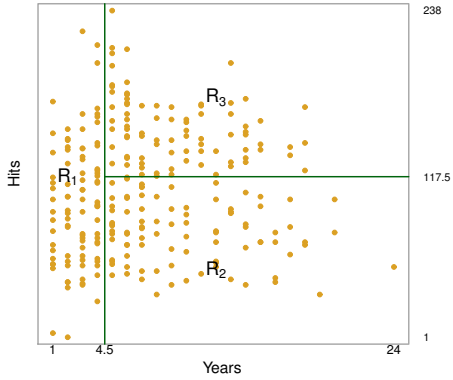
The Hitters salary tree, drawn as a flowchart



Takeaway

Each split is a yes/no question; each leaf reports the mean log-salary there. The highest earners sit in the deep right branch: experienced players with many hits.

Hitters: how the tree partitions the data



Each leaf is a rectangle in (Years, Hits).

Source: Figure 8.2 — ISLP, James et al. (2023).

Why prune?

Recursive splitting with no stopping rule gives one leaf per training point — zero training error, pathological test error.

Cost-complexity pruning

Choose a tree T minimising:

$$\sum_{j=1}^{|T|} \sum_{i \in R_j} (y_i - \bar{y}_{R_j})^2 + \alpha |T|.$$

How to read this

Here $|T|$ is the number of leaves and $\alpha \geq 0$ the price paid per leaf: the first term rewards training fit, the $\alpha|T|$ term punishes size. Larger $\alpha \Rightarrow$ smaller pruned tree; choose α by cross-validation.

Classification trees: splitting rules

Same recursive splitting; predict majority class in each leaf. Two natural splitting criteria:

Gini and cross-entropy

$$G = \sum_k \hat{p}_{mk}(1 - \hat{p}_{mk}), \quad D = -\sum_k \hat{p}_{mk} \log \hat{p}_{mk}.$$

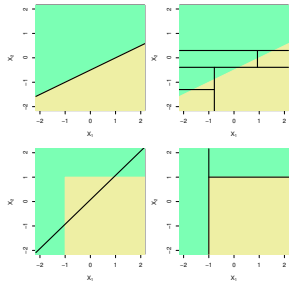
Reading the impurities

$\hat{p}_{mk}$ is the fraction of node-m points in class k. Both G and D are *small* when one class dominates ($\hat{p}_{mk} \approx 1$) and *large* at a 50/50 mix — so a good split makes the child nodes as *pure* as possible.

Why not error rate?

The misclassification rate has many ties on small samples — it splits poorly. Use Gini or cross-entropy for splitting; error rate for pruning.

Trees vs. linear models: when each shines



Takeaway

Top: with a linear boundary the linear model wins and the tree's box-steps lose. Bottom: with non-linear, interacting structure the tree wins.

Source: Figure 8.7 — ISLP, James et al. (2023).

Strengths and weaknesses of single trees

Strengths

- Easy to explain.
- Handle mixed and missing data.
- Capture interactions naturally.
- Robust to monotone transformations.

Weaknesses

- Unstable: small data changes $\rightarrow$ very different trees.
- Predictions are step functions $\rightarrow$ rough.
- Often beaten by linear models on simple problems.

In industry — Governance: the tree that fits on one page

A small pruned tree survives in governance-heavy settings precisely because it is weak but printable: underwriting triage, call-centre routing and credit policy overlays are often shipped as a tree a committee can read and sign. Grow it too deep and it stops being a rule set — and starts being unstable.

Exercise 8.1 — Regression tree split by RSS [Math]

Exercise

A regression tree is grown on this dataset:

x	1	2	3	4	5	6
y	1.0	1.5	2.0	8.0	8.5	9.0

1. List the candidate split points (midpoints between consecutive x).
2. Compute the total RSS for the splits $x < 3.5$ and $x < 2.5$.
3. Which split is chosen? Give the predictions for $x = 2$ and $x = 5$.
4. Predict y for a new point $x = 3.2$.

Solution — Exercise 8.1 (1/2)

Solution

(a) Candidate splits. Any cutpoint between two consecutive x-values produces the same partition, so the midpoints are the canonical candidates: $s \in \{1.5,\ 2.5,\ 3.5,\ 4.5,\ 5.5\}$.

(b) RSS per split. In each region the RSS-minimising constant prediction is the region *mean*; the RSS adds the squared deviations from it.
Split $x < 3.5$*:* left $\{1.0, 1.5, 2.0\}$, mean 1.5; right $\{8.0, 8.5, 9.0\}$, mean 8.5.

$$\mathrm{RSS}_L = (-0.5)^2 + 0^2 + 0.5^2 = 0.5, \quad \mathrm{RSS}_R = 0.5, \quad \text{total} = 1.0.$$

Split $x < 2.5$*:* left $\{1.0, 1.5\}$, mean 1.25; right $\{2.0, 8.0, 8.5, 9.0\}$, mean 6.875.

$$\mathrm{RSS}_L = (-0.25)^2 + 0.25^2 = 0.125,$$

$$\mathrm{RSS}_R = (-4.875)^2 + 1.125^2 + 1.625^2 + 2.125^2 \approx 32.19, \quad \text{total} \approx 32.31.$$

Solution — Exercise 8.1 (2/2)

Solution

(c) Chosen split and predictions. The split $x < 3.5$ gives total RSS 1.0 versus ≈ 32.31 for $x < 2.5$, so $x < 3.5$ is chosen — it separates the two tight y-clusters, whereas $x < 2.5$ leaves the outlying $y = 2.0$ stranded among the large values. Predictions are the leaf means: $x = 2$ lands left $\Rightarrow \hat{y} = 1.5$; $x = 5$ lands right $\Rightarrow \hat{y} = 8.5$.

(d) New point. $x = 3.2 < 3.5$, so it falls in the left region: $\hat{y} = 1.5$.

Interpretation: a regression tree predicts a *step function* — every point in a leaf gets the same value, no matter how close it sits to the boundary.

Common mistake

Interpolating between the two leaf means for a point near the cut (e.g. predicting ≈ 5 at $x = 3.2$). Trees do not interpolate; the prediction jumps only when x crosses 3.5.

Exercise 8.2 — Gini index vs. classification error [Math]

Exercise

A binary node contains 400 positive and 400 negative cases (800 total). Two candidate splits give:

- **Split 1:** region A $(300+, 100-)$ and region B $(100+, 300-)$.
- **Split 2:** region A $(200+, 400-)$ and region B $(200+, 0-)$.

1. Compute the weighted classification error for each split.
2. Compute the weighted Gini index for each split.
3. Which split does Gini prefer, and why is that desirable?
4. Why are Gini / entropy preferred for *growing* the tree?

Solution — Exercise 8.2 (1/2)

Solution

Set-up. For a node with positive proportion $\hat{p}$: error $E = 1 - \max(\hat{p}, 1-\hat{p})$ (the fraction misclassified by the majority vote) and Gini $G = 2\hat{p}(1-\hat{p})$. A split is scored by the *weighted* average of its two regions, each weighted by its share of the 800 cases — a large impure region must count more than a small one.

(a) Classification error. *Split 1:* each region has 400 cases with $\hat{p} = 300/400 = 0.75$, so $E = 1-0.75 = 0.25$ each; weighted:

$$\tfrac{400}{800}(0.25) + \tfrac{400}{800}(0.25) = 0.25.$$

Split 2: region A (600 cases) is majority negative, $E = 200/600 = 0.333$; region B (200 cases, pure) has $E = 0$. Weighted:

$$\tfrac{600}{800}(0.333) + \tfrac{200}{800}(0) = 0.25.$$

Both splits give error 0.25 — the error criterion finds them *indistinguishable*.

Solution — Exercise 8.2 (2/2)

Solution

(b) Gini. *Split 1:* each region $\hat{p} = 0.75$, $G = 2(0.75)(0.25) = 0.375$; weighted $= 0.375$. *Split 2:* region A $\hat{p} = 200/600 = 1/3$, $G = 2(\frac{1}{3})(\frac{2}{3}) = 0.444$, weight 0.75; region B pure, $G = 0$. Weighted $= 0.75(0.444) + 0.25(0) = 0.333$.

(c) Gini prefers **Split 2** ($0.333 < 0.375$) because it produces a *pure* node (region B is all positive). Node purity is valuable even when it does not change the majority-vote error rate: a pure node needs no further splitting and yields confident predictions. Entropy behaves the same way.

(d) Classification error is insensitive to node purity: it ignores improvements that do not flip the majority class. Gini and cross-entropy do reward purity, so they are used to *grow* the tree; classification error is adequate when judging (or pruning toward) the final tree's accuracy.

Common mistake

Averaging the two regions' impurities without their size weights — with unequal regions (Split 2) the unweighted average is misleading.

Exercise 8.3 — Cost-complexity pruning [Concept]

Exercise

Cost-complexity pruning selects a subtree $T \subset T_0$ minimising

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

1. What do $|T|$ and α represent?
2. What subtree results when $\alpha = 0$, and when $\alpha \to \infty$?
3. Why grow a large tree and then prune, rather than stop splitting early?
4. How is α chosen?

Solution — Exercise 8.3 (1/2)

Solution

(a) The two ingredients. $|T|$ is the number of terminal nodes (leaves) — the natural measure of tree complexity, since each leaf is one fitted constant. $\alpha \geq 0$ is a tuning parameter: the *price paid per leaf*. The criterion trades training fit (first term) against complexity ($\alpha|T|$), exactly as the lasso trades RSS against $\lambda \sum |\beta_j|$.

(b) The two extremes. $\alpha = 0$: leaves cost nothing, and every extra split can only lower the RSS term, so the full grown tree T_0 minimises the criterion. $\alpha \to \infty$: the penalty dominates, so the best subtree keeps as few leaves as possible — the *root alone*, a single node predicting the overall mean $\bar{y}$.

Interpretation: sliding α from 0 to ∞ walks through a family of subtrees from most to least complex — the tree analogue of a regularisation path.

Solution — Exercise 8.3 (2/2)

Solution

(c) Why grow-then-prune beats early stopping. Early stopping is short-sighted: a split that looks worthless now (tiny RSS reduction) may *enable* a very valuable split just below it — e.g. when two predictors matter only in combination, the first split shows little gain and a stopping rule would quit before the second, decisive split. Growing a large tree first and then pruning keeps such pairs, and cost-complexity pruning conveniently produces a *nested sequence* of subtrees indexed by α.

(d) Choosing α. By K-fold cross-validation: for a grid of α values, find the corresponding best subtree on each training fold, estimate its CV error, pick the α minimising CV error, then refit that subtree on the full data.

Common mistake

Selecting α on the *training* criterion — the RSS term always favours the biggest tree, so training data alone would pick $\alpha = 0$. Only held-out error can choose the penalty.

Extended Exercise 8.1 — Grow a regression tree by hand [Math]

Extended exercise (15 min)

Grow a regression tree on this data set (predictor x, response y):

x	1	2	3	4	5	6
y	3	4	12	13	20	21

1. Give the root prediction and its RSS.
2. Evaluate every candidate split (midpoints of consecutive x) and pick the one minimising total RSS.
3. Split the resulting right child once more.
4. State the final regions, their predictions, and the total RSS reduction.

Solution — Extended Exercise 8.1 (1/3)

Solution

(a) Root. Grand mean $\bar{y} = \frac{3+4+12+13+20+21}{6} = \frac{73}{6} \approx 12.17$, and

$$\mathrm{RSS}_{\text{root}} = \sum y_i^2 - 6\bar{y}^2 = 1179 - \frac{73^2}{6} \approx 290.83.$$

(b) Candidate splits. Predict each region by its mean; total RSS $= \mathrm{RSS}_L + \mathrm{RSS}_R$:

split	left | right mean	total RSS
$x < 1.5$	3.0 | 14.0	190.0
$x < 2.5$	3.5 | 16.5	65.5
$x < 3.5$	6.33 | 18.0	86.7
$x < 4.5$	8.0 | 20.5	82.5
$x < 5.5$	10.4 | 21.0	197.2

The split $x < 2.5$ minimises RSS.

Solution — Extended Exercise 8.1 (2/3)

Solution

First split $x < 2.5$. Left $\{3, 4\}$ (mean 3.5, RSS 0.5); right $\{12, 13, 20, 21\}$ (mean 16.5, RSS 65). The RSS drops from 290.83 to 65.5.

(c) Split the right child $\{x \geq 2.5\} = \{12, 13, 20, 21\}$ at $x = 3, 4, 5, 6$:

split	left | right mean	child RSS
$x < 3.5$	12.0 | 18.0	38.0
$x < 4.5$	12.5 | 20.5	1.0
$x < 5.5$	15.0 | 21.0	38.0

The split $x < 4.5$ wins, cutting the child's RSS from 65 to 1.0.

Common mistake

Evaluating the second split on *all six* points. Recursive splitting works within one node at a time — only the right child's four points $\{12, 13, 20, 21\}$ enter this computation.

Solution — Extended Exercise 8.1 (3/3)

Solution

(d) Final tree (three leaves).

region	rule	$\hat{y}$
R_1	$x < 2.5$	3.5
R_2	$2.5 \leq x < 4.5$	12.5
R_3	$x \geq 4.5$	20.5

Final total RSS $= 0.5 + 0.5 + 0.5 = 1.5$.

Total RSS reduction.

$$290.83 \xrightarrow{\text{split 1}} 65.5 \xrightarrow{\text{split 2}} 1.5,$$

a reduction of $290.83 - 1.5 = 289.33$ (225.33 from the first split, 64.0 from the second). The tree explains almost all variability because the data fall into three tight clusters.

Why stop here: each leaf's remaining RSS is only 0.5; further splits would chase noise and cost-complexity pruning (any $\alpha > 0.5$) would remove them again.

Outline

Bagging: variance reduction by averaging

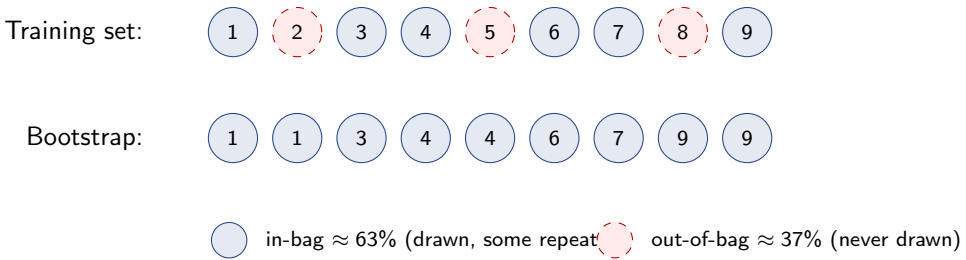
Algorithm

1. Draw B bootstrap samples of size n.
2. Fit a deep tree to each sample.
3. Average B predictions (regression) or majority-vote (classification).

Free benefits

- Variance reduction via averaging — bias unchanged.
- OOB error: each training point appears in $\sim 63\,\%$ of the bootstrap samples; the held-out $37\,\%$ act as built-in CV.
- No tuning required.

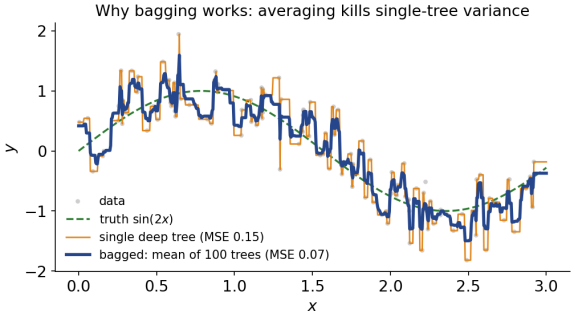
Out-of-bag: the leftover third is a free test set



Takeaway

A bootstrap draw of n points with replacement repeats some (in-bag) and misses others: $\left(1-\frac{1}{n}\right)^n \to e^{-1} \approx 0.37$. Scoring each point with only the trees that did *not* see it gives the OOB error — built-in cross-validation, for free.

Why averaging helps: one deep tree vs. a bagged hundred



Takeaway

On simulated $y = \sin(2x) + \varepsilon$ a single deep tree is a jagged step function (MSE vs. truth 0.15); averaging 100 bootstrap trees smooths it to MSE 0.07 — bagging averages away single-tree variance while the bias stays put.

Random forests: the de-correlation trick

Bagging averages *correlated* trees because the same strong predictor tends to dominate the top split.

Key idea

At each split, consider only m randomly chosen predictors out of p.

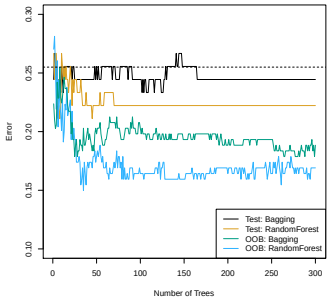
Default values of m

- Classification: $m = \sqrt{p}$.
- Regression: $m = p/3$.

Takeaway

Forcing different trees to use different predictors lowers their pairwise correlation, which makes the average more effective. With $m = p$ random forests reduce to bagging.

Random forest: error vs. number of trees

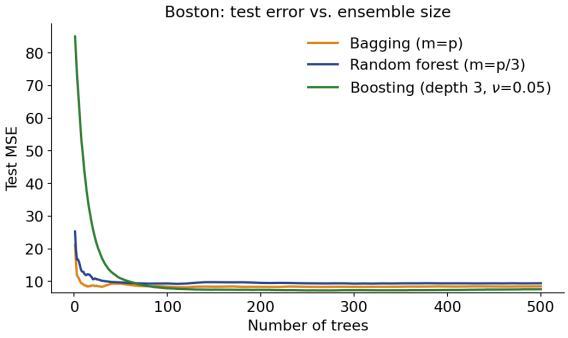


Takeaway

Test error falls then flattens after a few hundred trees. More trees never hurt — unlike boosting, you cannot "overfit" a forest by growing it larger, only hit diminishing returns.

Source: Figure 8.8 — ISLP, James et al. (2023).

Test error vs. ensemble size: bagging, forest, boosting



Takeaway

All three ensembles beat a single tree. Bagging and the forest fall fast then flatten; slow-learning boosting starts higher but reaches the lowest test MSE.

Variable importance in random forests

Two notions

- **Impurity importance**: total decrease in node impurity due to splits on X_j, summed across trees.
- **Permutation importance**: shuffle X_j in OOB samples and measure the increase in error.

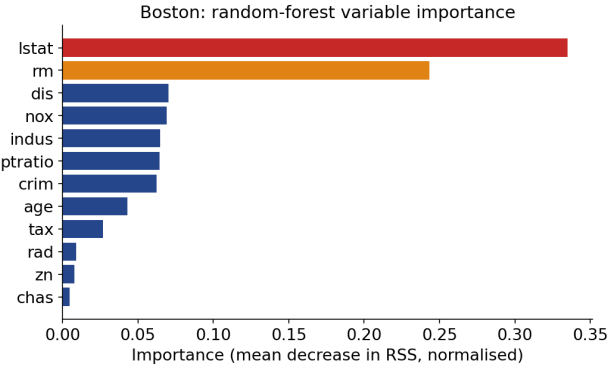
Caveat

Correlated predictors share importance — you may need to interpret pairs together.

In industry — Model risk: an importance ranking is not an explanation

Impurity importance is biased towards high-cardinality variables (customer ID, postcode) and correlated predictors split the credit, so risk teams report permutation importance or SHAP instead — the same attributions that produce the reason codes a declined applicant must be given.

Variable importance on the Boston data



Takeaway

Two predictors dominate: `lstat` (% lower-status population) and `rm` (average rooms). Housing value is driven mostly by neighbourhood status and dwelling size; the rest contribute little.

Mini-check: trees and ensembles

Quick self-assessment

1. Why is a single tree typically high-variance?
2. Why does feature subsampling at each split improve random forests over plain bagging?
3. Can you "overfit" a random forest by growing too many trees?

Answers

1. Greedy splits, no shrinkage, perfectly fit training data. 2. Decorrelates the trees, so averaging is more effective. 3. No — adding more trees never hurts, you just hit diminishing returns.

Exercise 8.4 — Bagging and OOB error [Concept]

Exercise

1. If each tree has variance σ^2, what is the variance of the average of B trees when they are independent? When they have pairwise correlation ρ?
2. What fraction of observations are out-of-bag (OOB) for a given bootstrap tree? Derive it.
3. What does the OOB error estimate, and why is it almost free?

Solution — Exercise 8.4 (1/2)

Solution

(a) Variance of the average. For the mean of B trees, expand the variance of a sum (B variance terms, $B(B-1)$ covariance terms $\rho\sigma^2$):

$$\mathrm{Var}\Big(\tfrac{1}{B}\sum_b \hat{f}_b\Big) = \frac{1}{B^2}\Big[B\sigma^2 + B(B-1)\rho\sigma^2\Big] = \rho\sigma^2 + \frac{1-\rho}{B}\,\sigma^2.$$

Independent trees ($\rho = 0$): $\mathrm{Var} = \sigma^2/B \to 0$ — averaging could remove *all* variance. *Correlated trees:*

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \xrightarrow{B\to\infty} \rho\sigma^2.$$

Interpretation: bagging drives the second term to zero for free (more trees), but the floor $\rho\sigma^2$ remains — the correlation between trees caps how much variance averaging can remove. This is exactly the lever random forests attack.

Solution — Exercise 8.4 (2/2)

Solution

(b) OOB fraction. A bootstrap sample makes n independent draws *with replacement*; each draw misses observation i with probability $1 - \frac{1}{n}$, so

$$\Pr(i \text{ never drawn}) = \left(1 - \tfrac{1}{n}\right)^n \xrightarrow{n\to\infty} e^{-1} \approx 0.368$$

(already 0.366 at $n = 100$). So about $1/3$ of observations are OOB and $2/3$ in-bag for each tree.

(c) OOB error. Each observation is OOB for roughly $B/3$ of the trees; predicting it using *only those* trees gives an honest held-out prediction. Aggregating over all n observations yields the OOB error, which approximates the test / CV error — and it comes free with a single fit, since it reuses trees that were grown anyway.

Common mistake

Scoring OOB points with *all* B trees. Trees that saw the point in-bag have memorised it, so including them makes the error estimate optimistically low.

Exercise 8.5 — Random forests and decorrelation [Concept/Math]

Exercise

1. How does a random forest differ from bagging? What is m?
2. Give the usual choice of m for classification and for regression with p predictors.
3. Using the variance formula $\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$, explain why restricting to m predictors helps.
4. With $p = 16$, what is m for classification and for regression?

Solution — Exercise 8.5 (1/2)

Solution

(a) The difference. A random forest is bagging plus a twist: at *each split* only a random subset of m of the p predictors is considered as candidates. The subset is redrawn *at every split*, not once per tree, so a single tree can still use all predictors — just never all at the same decision. Bagging is the special case $m = p$.

(b) Default values. Classification: $m \approx \sqrt{p}$. Regression: $m \approx p/3$. These are rules of thumb — m is a tuning parameter worth checking by OOB error or CV.

Interpretation: m is the decorrelation dial: $m = p$ recovers bagging, small m forces diverse trees.

Solution — Exercise 8.5 (2/2)

Solution

(c) Why restricting to m predictors helps. If one or two predictors are very strong, in bagging almost every bootstrap tree splits on them at the top, so the trees look alike — ρ is large and the limiting variance $\rho\sigma^2$ stays high, no matter how big B is. With $m < p$ a split ignores each given predictor with probability $(p - m)/p$ — e.g. $m = 4$ of $p = 16$ hides the dominant predictor at $12/16 = 75\,\%$ of splits — so trees are forced to differ. That lowers ρ, hence the floor $\rho\sigma^2$, and averaging becomes far more effective.

(d) With $p = 16$: classification $m = \sqrt{16} = 4$; regression $m = 16/3 \approx 5$ (round 5.33 down to an integer).

Common mistake

Expecting small m to reduce *bias*. Restricting candidates slightly *raises* each tree's bias; the payoff is purely in decorrelation, i.e. variance reduction of the average.

Outline

Boosting: idea

Boosting builds a sequence of weak trees, each fitted to the residuals of the previous fit. Slow, careful adaptation.

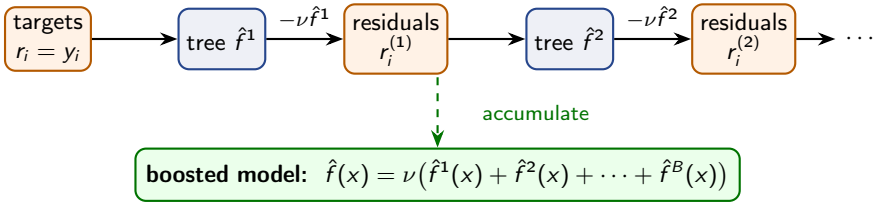
Gradient-boosting algorithm

1. Initialise $\hat{f}(x) = 0$, $r_i = y_i$.
2. For $b = 1, \ldots, B$:
   1. Fit a tree $\hat{f}^b$ with d splits to (x_i, r_i).
   2. Update $\hat{f} \leftarrow \hat{f} + \nu \hat{f}^b$.
   3. Update residuals $r_i \leftarrow r_i - \nu \hat{f}^b(x_i)$.

In industry — Banking: champion and challenger

Many lenders keep an interpretable logistic scorecard as the model of record for the regulated decision and run a boosted model alongside it as a challenger, on discrimination and stability. The boosted model usually wins on accuracy; whether it may take the decision is a model-governance question, not a statistical one.

Boosting: each tree fits what the last one missed



Reading the update

Fit a small tree to the current *residuals* r_i (what the model still gets wrong), add only a fraction ν of it ($0<\nu\ll1$), then shrink the residuals. Small ν = slow, cautious learning; the depth d caps the interaction order.

Boosting tuning parameters

The trio

- B — number of trees. Too large $\Rightarrow$ overfitting.
- ν — learning rate (typically 0.01 or 0.001).
- d — tree depth (often $d = 1$ or 2 suffices).

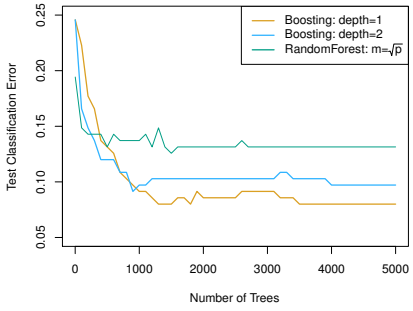
Trade-off

Smaller ν usually wins but requires larger B. The product $\nu \cdot B$ controls effective fitting effort. Choose all three by CV.

In industry — Forecasting and fraud: validate by time — never at random

In production, B is fixed by early stopping on a held-out period. The random K-fold CV of Chapter 5 leaks the future into training on transaction or sales data and flatters the model; teams validate on later weeks than they train on.

Boosting vs. random forest: head-to-head



Takeaway

With enough trees and a small learning rate, boosting often edges out the random forest on test error — but it must be tuned carefully, whereas the forest is strong out of the box.

Source: Figure 8.11 — ISLP, James et al. (2023).

Modern boosting libraries

Industrial implementations dominate Kaggle on tabular data.

The big three

`XGBoost`, `LightGBM`, `CatBoost`.

Beyond textbook gradient boosting

Feature subsampling, second-order gradients, ℓ_1 / ℓ_2 regularisation on leaf weights, GPU support, native categorical handling, histogram-based splits.

In industry — Production: why these libraries are the default

Native handling of missing values and categorical variables removes most feature engineering, and histogram splits make millions of rows routine — which is why the same three libraries sit behind demand forecasts, fraud scores and ad ranking. Cost of ownership: scoring latency, retraining cadence, drift monitoring.

Exercise 8.6 — Boosting tuning parameters [Concept]

Exercise

Gradient boosting with regression trees has three tuning parameters.

1. Name them and state what each controls.
2. What does "slow learning" mean and why does it help?
3. How do B and λ interact, and what is the danger of a large B?
4. What does d control? What model do we get when $d = 1$?

Solution — Exercise 8.6 (1/2)

Solution

(a) The three parameters. B = number of trees (how many boosting rounds); λ = shrinkage / learning rate (typically 0.01 or 0.001) — the fraction of each new tree actually added; d = number of splits per tree (interaction depth / tree size).

(b) Slow learning. Each new tree is fitted to the current *residuals* — what the ensemble still gets wrong — but only λ times its prediction is added:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x), \qquad r_i \leftarrow r_i - \lambda \hat{f}^b(x_i).$$

With small λ no single tree can dominate; the ensemble creeps toward the signal in many tiny, mutually correcting steps rather than a few large ones. This shrinkage reduces overfitting and improves generalisation — the same logic as ridge/lasso shrinkage, applied to trees.

Solution — Exercise 8.6 (2/2)

Solution

(c) How B and λ interact. A smaller λ requires a larger B to reach the same total fitting effort (roughly, effort $\propto \lambda \cdot B$) — they trade off. The danger: unlike random forests, boosting keeps re-fitting the residuals, so with B too large it starts fitting *noise* — it *can* overfit. Choose B by cross-validation or early stopping on a validation set.

(d) The role of d. d caps how many variables can interact along one tree's path, i.e. the *interaction order*. $d = 1$ gives stumps: each tree involves a single predictor, so the boosted model is purely *additive* — a tree-built cousin of a GAM. Larger d allows higher-order interactions among predictors.

Common mistake

Tuning B by training error — it decreases monotonically in B, so it never signals when to stop; only validation/CV error turns upward.

Outline

Python lab — run it live in the notebook

Companion notebook: `chapter_08_lab.ipynb`

The code in this section is meant to be **demonstrated live**. Open the companion Jupyter notebook and run it cell by cell, section by section:

- §2, *Single regression tree*, and §3, *Pruning via cost-complexity* — on the Carseats data loaded in §1;
- §4, *Random forest*, and §5, *Gradient boosting*;
- the *Optional: XGBoost* subsection that closes §5.

Data loads via the `ISLP` package or the bundled CSV files; §6, *Exercises*, closes the notebook.

Decision tree on Carseats

```
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.model_selection import train_test_split
from ISLP import load_data

Carseats = load_data("Carseats") # load data
X = pd.get_dummies(Carseats.drop(columns="Sales"), drop_first=True) # dummy-code factors
y = Carseats["Sales"] # response

Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, random_state=0) # 70/30 split
tree = DecisionTreeRegressor(max_depth=4).fit(Xtr, ytr) # grow a depth-4 tree
plot_tree(tree, feature_names=X.columns, filled=True, fontsize=7) # draw fitted tree
```

Random forest

```
from sklearn.ensemble import RandomForestRegressor
import numpy as np

rf = RandomForestRegressor(n_estimators=500, max_features="sqrt", # B=500, m=sqrt(p)
                           random_state=0, oob_score=True).fit(Xtr, ytr) # keep OOB score
print("OOB R^2:", rf.oob_score_) # OOB error: free validation
print("Test MSE:", ((yte - rf.predict(Xte))**2).mean()) # held-out test error

imp = pd.Series(rf.feature_importances_, index=X.columns) # impurity importances
imp.sort_values().plot.barh() # importance plot
```

Gradient boosting and XGBoost

```
from sklearn.ensemble import GradientBoostingRegressor
gb = GradientBoostingRegressor(n_estimators=1000, learning_rate=0.01, # B=1000, lambda=0.01
                               max_depth=3, random_state=0).fit(Xtr, ytr) # depth d=3
print("Test MSE:", ((yte - gb.predict(Xte))**2).mean()) # boosted test error

# Optional XGBoost
import xgboost as xgb
bst = xgb.XGBRegressor(n_estimators=2000, learning_rate=0.05, # more trees, larger step
                       max_depth=4, subsample=0.8).fit(Xtr, ytr) # row subsampling
```

Exercise 8.7 — Random forest & boosting on Boston [Python]

Exercise

Using `Boston.csv` (target `medv`):

1. Fit a random forest and a gradient-boosting regressor; report test MSE.
2. Read off the most important features from the random forest.
3. Interpret the result.

Solution — Exercise 8.7 (1/3)

Solution

```
import pandas as pd
from sklearn.ensemble import (RandomForestRegressor,
                              GradientBoostingRegressor)
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

Boston = pd.read_csv('../ALL CSV FILES - 2nd Edition/Boston.csv')
Boston = Boston.drop(columns=[c for c in ['Unnamed: 0'] # drop index col
                              if c in Boston.columns])
X = Boston.drop(columns='medv'); y = Boston['medv'] # target: medv
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, # 70/30 split
                                      random_state=0)
```

Set-up: we hold out 30 % as a test set; fixing `random_state` makes the split (and every number below) reproducible.

Solution — Exercise 8.7 (2/3)

Solution

```
rf = RandomForestRegressor(n_estimators=500, max_features='sqrt', # 500 trees, m=sqrt(p)
                           random_state=0).fit(Xtr, ytr) # fit on training set
print('RF test MSE:', mean_squared_error(yte, rf.predict(Xte))) # held-out error

gb = GradientBoostingRegressor(n_estimators=1000, learning_rate=0.01, # slow learning
                               max_depth=3, random_state=0).fit(Xtr, ytr) # depth 3
print('GBM test MSE:', mean_squared_error(yte, gb.predict(Xte))) # compare with RF

imp = pd.Series(rf.feature_importances_, # impurity importances
                index=X.columns).sort_values(ascending=False) # largest first
print(imp.head()) # top predictors
```

What it prints (this seed): RF test MSE ≈ 19.6, boosting ≈ 13.5 — versus ≈ 27 for a single unpruned tree on the same split. The importance list is headed by `lstat` (≈ 0.28) and `rm` (≈ 0.27).

Solution — Exercise 8.7 (3/3)

Solution

Interpretation. Both ensembles reach a test MSE well below a single tree (typically around 10–15 for Boston). On this particular split the $m = \sqrt{p}$ forest lands higher (≈ 19.6) while boosting reaches ≈ 13.5: Boston has a few dominant predictors, so restricting each split to $\sqrt{12} \approx 3$ candidates costs accuracy here — Extended Exercise 8.3 shows that a larger m closes the gap.

The importance ranking is dominated by `lstat` (% lower-status population) and `rm` (average number of rooms): housing value depends most on socio-economic status and dwelling size. Importances measure the total RSS reduction attributable to each variable summed across all trees, normalised to sum to 1.

Common mistake

Comparing models fitted or evaluated on *different* splits — always fix one train/test split (or use CV) before declaring a winner, and remember that single-split MSEs carry noticeable seed-to-seed noise.

Extended Exercise 8.3 — Bagging vs. forest vs. boosting [Python]

Extended exercise (15 min)

Using `Boston.csv` (target `medv`), fit and compare three ensembles against a single tree:

1. **Bagging** (a forest that considers all p predictors at each split);
2. a **random forest**, tuning $m =$ features tried per split;
3. **gradient boosting**, tuning $(B, \lambda, d) =$ (trees, learning rate, depth).

Report the test MSE of each, and read off the random-forest variable-importance ranking. Which method wins, and which predictors matter?

Run this live: *Extended Exercise 8.3 — Bagging vs. random forest vs. boosting*

Under *Lecture exercises — worked Python solutions* in `chapter_08_lab.ipynb`: all four fits on one split, with the importance ranking, ready to run.

Solution — Extended Exercise 8.3 (1/3)

Solution

Load the data; fit a single tree and bagging (max_features=p = bagging).

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_squared_error as mse
B = pd.read_csv('../ALL CSV FILES - 2nd Edition/Boston.csv') # load
B = B.drop(columns=[c for c in ['Unnamed: 0', ''] if c in B.columns])
X, y = B.drop(columns='medv'), B['medv']; p = X.shape[1] # p=12
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=.3, random_state=1)
tree = DecisionTreeRegressor(random_state=1).fit(Xtr, ytr) # deep tree
bag = RandomForestRegressor(n_estimators=500, max_features=p, # m=p
                            random_state=1).fit(Xtr, ytr) # bagging
print('tree', round(mse(yte, tree.predict(Xte)), 2), # compare test MSE
      'bag', round(mse(yte, bag.predict(Xte)), 2))
```

What it prints (seed 1): tree ≈ 26.7, bagging ≈ 8.5 — averaging 500 bootstrap trees alone cuts the single tree's test error to a third.

Solution — Extended Exercise 8.3 (2/3)

Solution

Tune the forest's m and boosting's (B, λ, d); read importances.

```
for m in ['sqrt', 4, 6]: # tune random-forest m
    rf = RandomForestRegressor(n_estimators=500, max_features=m,
                               random_state=1).fit(Xtr, ytr)
    print('RF m=', m, round(mse(yte, rf.predict(Xte)), 2)) # error per m

for lr, Bn, d in [(0.1, 1000, 2), (0.01, 2000, 3)]: # tune B, lambda, d
    gb = GradientBoostingRegressor(learning_rate=lr, n_estimators=Bn,
                                   max_depth=d, random_state=1).fit(Xtr, ytr)
    print('GBM', lr, Bn, d, round(mse(yte, gb.predict(Xte)), 2)) # error

imp = pd.Series(rf.feature_importances_, # last RF's importances
                index=X.columns).sort_values(ascending=False)
print(imp.round(3).head()) # top predictors
```

What it prints (seed 1): RF $m{=}\sqrt{p}$: 9.79, $m{=}4$: 9.47, $m{=}6$: 8.90; GBM $(0.1, 1000, 2)$: 10.24, $(0.01, 2000, 3)$: 7.36 — slow learning with more trees wins.

Solution — Extended Exercise 8.3 (3/3)

Solution

Typical test MSE (with `random_state=1`; exact figures vary):

single tree	≈ 27
bagging ($m = p$)	≈ 8.5
random forest ($m = \sqrt{p} \;\rightarrow\; 6$)	$\approx 9.8 \;\rightarrow\; 8.9$
boosting (λ=0.01, B=2000, d=3)	≈ 7.4

Ranking: every ensemble slashes the single tree's error 3–4-fold; well-tuned *boosting* edges out bagging and the forest. Because Boston has few, strong predictors, a larger m helps, so bagging is competitive with the forest here. **Importance:** `lstat` (≈ 0.40) and `rm` (≈ 0.27) dominate, then `dis`, `nox` — housing value is driven mostly by neighbourhood socio-economic status and dwelling size. Boosting wins on accuracy but needs tuning; the forest is a strong low-effort default.

Common mistake

Reading importances as causal effects. They measure how much the ensemble *uses* a variable, and correlated predictors share the credit — `lstat`'s 0.40 partly stands in for the whole socio-economic cluster.

Outline

Chapter 8 in one slide

What we accomplished

The dominant family of methods on tabular data: trees and their ensembles.

- Recursive partitioning, pruning, classification trees.
- Bagging averages many trees to cut variance.
- Random forests decorrelate trees by random feature subsetting.
- Boosting fits sequential weak learners to residuals.
- BART: a Bayesian sum of small trees.

Key formulas at a glance

Regression tree minimise $\sum_m \sum_{i \in R_m} \left(y_i - \hat{y}_{R_m}\right)^2$; predict region mean.

Gini $G = \sum_k \hat{p}_{mk}(1 - \hat{p}_{mk})$; **entropy** $D = -\sum_k \hat{p}_{mk} \log \hat{p}_{mk}$.

Pruning minimise $\sum_{m=1}^{|T|} \mathrm{RSS}_m + \alpha |T|$ (cost complexity, $\alpha \geq 0$).

Bagging $\hat{f}_{\mathrm{bag}}(x) = \dfrac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$; OOB error $\approx$ test error.

Averaging gain $\mathrm{Var}\left(\frac{1}{B}\sum_b \hat{f}_b\right) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \xrightarrow{B \to \infty} \rho\sigma^2$ (each $\mathrm{Var} = \sigma^2$, pairwise corr. ρ).

Random forest as bagging but try $m \approx \sqrt{p}$ (class.) or $p/3$ (regr.) predictors per split.

Boosting $\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$; tune B, λ, d (slow learning).

Vocabulary checklist

Term	Meaning
Recursive binary splitting	Greedy node-by-node tree growth
Cost-complexity pruning	Penalise tree size by $\alpha\|T\|$
Gini / cross-entropy	Impurity for classification splits
Bagging	Average B bootstrap-trained trees
OOB error	Out-of-bag prediction error
Random forest	Bagging + random feature subsetting
Boosting	Sequential additive trees on residuals
Learning rate ν	Shrinkage applied to each boosting step
BART	Bayesian sum of trees
Variable importance	Total impurity / accuracy decrease per predictor

Decision rules of thumb

- Single tree: only as a baseline or for max interpretability.
- Random forest: an excellent default with very little tuning.
- Boosting: typically best accuracy, tune ν, depth, B jointly.
- Use OOB error in random forests — it's free CV.
- Permutation importance is more honest than impurity importance.
- Trees handle missing data and mixed types out of the box.

Common pitfalls

Don't

1. Use very deep boosted trees with high ν — they overfit.
2. Trust importance rankings under heavy collinearity.
3. Compare a tuned RF to an untuned linear model and conclude RF always wins.
4. Forget that trees fit step-functions — they cannot extrapolate.
5. Skip CV: even RF can overfit if individual trees are grown to unlimited depth.

Self-check questions

Each question names the slide that answers it — a wrong answer tells you where to read.

1. Why is recursive binary splitting greedy? When does this hurt? (p. 16)
2. Why use cross-entropy or Gini, not the misclassification rate, to grow classification trees? (p. 21)
3. Explain why random forests reduce variance more than plain bagging. (p. 41)
4. Walk through the gradient-boosting algorithm in pseudocode. (p. 54)
5. In what sense is BART a Bayesian boosting algorithm? (appendix, p. 90)
6. Rank baseline methods (logistic, RF, gradient boosting) for tabular data with $p = 100$, $n = 1000$. (p. 79)

Eight things to remember

1. Trees split predictor space into rectangles.
2. Greedy recursive binary splitting; cost-complexity pruning.
3. Single trees are interpretable but high variance.
4. Bagging: variance reduction by averaging.
5. Random forests: bagging + feature subsampling.
6. Boosting: sequential fit to residuals.
7. BART: Bayesian sum of small trees.
8. Tree ensembles dominate tabular ML.

What's next

Chapter 10: *Deep Learning*.

- Multilayer perceptrons (MLPs).
- Convolutional networks (CNNs) for images.
- Training: backpropagation, SGD, regularisation.

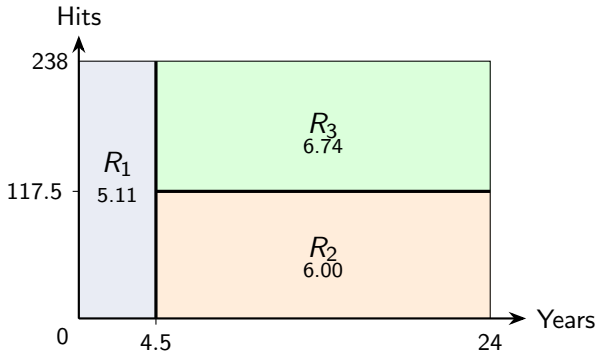
Appendix — what is in here, and why

Nothing in the appendix is needed to follow the chapter: each slide is a formal derivation, a longer worked exercise, or a side topic. Read it when you want the full story, or when a later chapter sends you back here.

Contents of the appendix

- **The partition picture, drawn a second time** — a schematic duplicate of the Hitters partition already shown in the main deck.
- **Impurity measures and pruning (Extended Exercise 8.2)** — Gini, entropy and cost-complexity pruning worked through in full. Exercises 8.2 and 8.3 cover both in the main deck.
- **BART** — Bayesian additive regression trees — a fourth ensemble, listed for completeness. Random forests and boosting are the two you will be asked about.

The same tree as a partition of predictor space



Takeaway

R_1 holds inexperienced players; among experienced players, more hits (R_3) predicts higher salary than fewer (R_2).

Extended Exercise 8.2 — Impurity measures and pruning [Math]

Extended exercise (15 min)

A binary-classification node holds 200 cases: 100 "Yes" and 100 "No". Two candidate splits give:

- **Split A:** $A_1 = (80\text{ Yes}, 20\text{ No})$ and $A_2 = (20\text{ Yes}, 80\text{ No})$.
- **Split B:** $B_1 = (100\text{ Yes}, 40\text{ No})$ and $B_2 = (0\text{ Yes}, 60\text{ No})$.

1. For each split compute the weighted Gini index, cross-entropy and classification error.
2. Which split wins under each criterion? Explain the disagreement.
3. A pruned-tree sequence has subtrees with (leaves $|T|$, training errors R) $= (4, 0), (3, 4), (2, 10), (1, 40)$. For $\alpha \in \{0, 5, 20, 50\}$ find the subtree minimising $R + \alpha|T|$.

Solution — Extended Exercise 8.2 (1/3)

Solution

For a node with "Yes"-fraction $\hat{p}$:

$$G = 2\hat{p}(1-\hat{p}), \quad D = -\hat{p}\ln\hat{p} - (1-\hat{p})\ln(1-\hat{p}), \quad E = 1 - \max(\hat{p}, 1-\hat{p}),$$

and children are combined by their share of the 200 cases.

Split A (A_1, A_2 each 100 cases, $\hat{p} = 0.8$ and 0.2 — symmetric):

$$G = 2(0.8)(0.2) = 0.32, \quad D = -0.8\ln 0.8 - 0.2\ln 0.2 = 0.500, \quad E = 0.20.$$

Both children give identical values and equal weights $\frac{100}{200} = 0.5$, so $G_A = 0.32$, $D_A = 0.500$, $E_A = 0.20$.

Note: we use the natural log for D; switching to $\log_2$ rescales every entropy by the same factor and never changes which split wins.

Solution — Extended Exercise 8.2 (2/3)

Solution

Split B. B_1: $\hat{p} = 100/140 = 0.714$, weight 0.7; B_2 pure ($\hat{p} = 0$), weight 0.3.

$$G_B = 0.7\,[2(0.714)(0.286)] + 0.3\,(0) = 0.7(0.408) = 0.286,$$
$$D_B = 0.7\,[-0.714 \ln 0.714 - 0.286 \ln 0.286] + 0 = 0.7(0.598) = 0.419,$$
$$E_B = 0.7\,(1 - 0.714) + 0.3\,(0) = 0.7(0.286) = 0.20.$$

(2) Who wins? Classification error ties ($E_A = E_B = 0.20$) and cannot separate the splits. Gini ($0.286 < 0.32$) and entropy ($0.419 < 0.500$) both prefer **Split B**, because it carves out a *pure* node B_2. Purity is rewarded by Gini/entropy even when majority-vote error is unchanged — which is why we *grow* trees with Gini or entropy, not error.

Solution — Extended Exercise 8.2 (3/3)

Solution

(3) Cost-complexity pruning. Minimise $R_\alpha(T) = R(T) + \alpha|T|$ over the nested subtrees:

	$(4,0)$	$(3,4)$	$(2,10)$	$(1,40)$
α=0	0	4	10	40
α=5	20	19	20	45
α=20	80	64	50	60
α=50	200	154	110	90

Winners: α=0 $\to$ 4 leaves, α=5 $\to$ 3 leaves, α=20 $\to$ 2 leaves, α=50 $\to$ the root. As α grows the optimal subtree shrinks (breakpoints at $\alpha = 4, 6, 30$) — the nested "weakest-link" sequence that cross-validation then searches over.

Sample entry: for $\alpha = 5$ and the 3-leaf tree, $R_\alpha = R + \alpha|T| = 4 + 5 \cdot 3 = 19$ — the smallest value in its row, so 3 leaves win at that price.

Common mistake: choosing α by training error — $\alpha = 0$ (the full tree) always wins there, by construction. α is chosen by cross-validating over the pruning sequence.

BART: Bayesian Additive Regression Trees

Idea

A sum of K small trees (each a "weak learner"), updated one tree at a time using a Bayesian posterior-style move.

Takeaway

Combines flavours of bagging (averaging) and boosting (sequential update). Out-of-the-box hyperparameters often work; the Bayesian framework gives credible intervals for free. Implementations: `ISLP.bart`, `PyMC`.