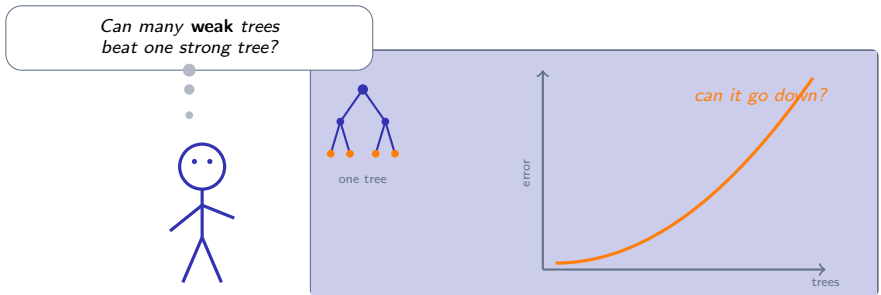
XGBoost: A Simple, Complete Lecture

From Gradient Boosting to the Kaggle Gold Standard – BSc Data Science

60 Minutes



Gradient boosting: small corrections, one step at a time.

Why XGBoost?

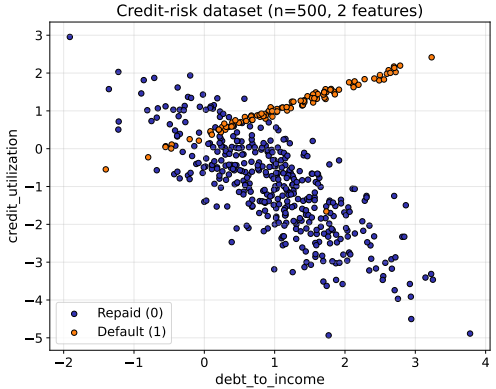
Plain English: XGBoost has dominated tabular-data prediction since 2014. If your data is rows in a spreadsheet, this is the first thing to try.

Where it shows up.

- **Banks:** credit scoring, default prediction
- **Insurers:** fraud detection, claim cost
- **Kaggle:** top-3 finisher on tabular contests for a decade

Why it wins.

- Fast (parallel split-finding)
- Accurate (gradient boosting + regularization)
- Handles missing values out of the box



Tabular data ⇒ XGBoost first. The runway pattern from the last decade.

Our Playground: a Credit-Risk Dataset

500 synthetic borrowers, 2 features.

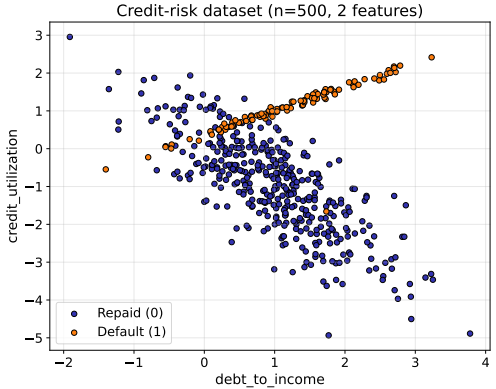
- `debt_to_income` – monthly debt / monthly income
- `credit_utilization` – balance / credit limit

Two classes.

- **Class 0 = repaid** (purple, 80%)
- **Class 1 = default** (orange, 20%)

Standard split. 70% train / 30% test, `random_state=42`.

Every decision-boundary chart in this deck uses this 2D dataset. The feature-importance + SHAP charts (slides 42–43) use a sibling 6-feature set.



One simple 2D playground keeps every later chart comparable.

What Is a Decision Tree? Recap.

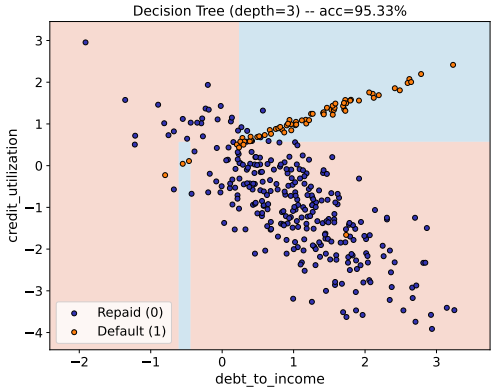
Plain English: A **decision tree** is a *flowchart*: each *internal node* asks "is feature $X \leq$ threshold?"; each *leaf* (terminal node) holds one prediction. Start at the root; follow branches until a leaf.

Mechanics.

- Each internal node: *one feature, one threshold* (the *threshold* is the cut value that divides data)
- Each leaf: terminal node that holds one prediction (class or value)
- Each *split* divides data along one feature at one threshold
- `max_depth=3` $\Rightarrow$ at most $2^3 = 8$ leaves

Strengths and weaknesses.

- Strength: fast, easy to explain
- Weakness: *unstable* – small data changes flip the boundary



One tree is fast but brittle. The next slides combine many of them.

Anatomy of a Tree (Picture + Words)

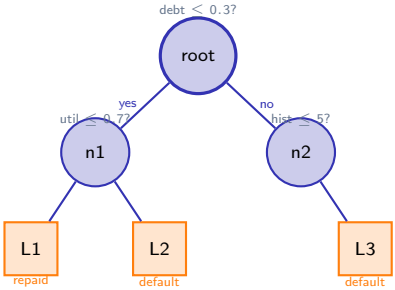
Three kinds of node.

- **Root** – top of the tree, starting point (one node)
- **Internal node** – asks one yes/no question on one feature with a threshold
- **Leaf** – terminal node, holds one prediction (no further question)

An example rule path.

- Root: "debt ≤ 0.3?"
- If yes, internal node 2: "utilisation ≤ 0.7?"
- If yes, leaf: predict "repaid" ($w = 0.1$)

Read the picture as a sequence of yes/no questions.



Key terms. root (purple), internal node (purple circle), leaf (orange box).

Anatomy: root → internal nodes → leaves. Each path is a chain of yes/no questions.

What's a Stump?

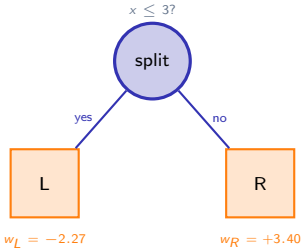
Definition. A **stump** is a decision tree of *depth 1* – one split, two leaves, nothing more.

Why so small?

- One internal node (the split), one threshold on one feature
- Two leaves (left and right of the threshold)
- Cannot capture interactions between features
- Predictions are constant within each half

Why care? Boosting prefers *shallow* trees. Stumps are the simplest weak learner that still does something useful.

In Worked Example A, we will fit exactly one stump per iteration.



Stump = depth 1 = one split = two leaves.

This is the workhorse of boosting. A stump is too weak alone; in series, hundreds of them become a strong model.

Stump = depth-1 tree. One split, two leaves. Boosting's favourite weak learner.

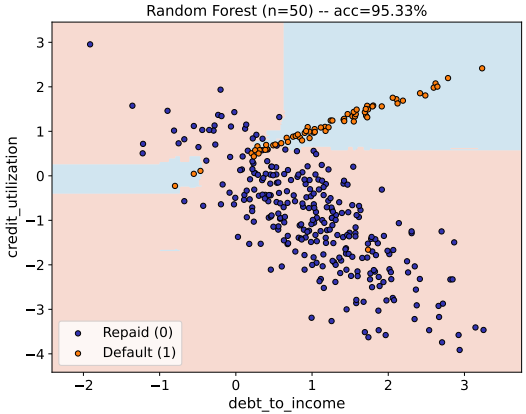
From One Tree to Many Trees

Bagging (Random Forest).

- Many trees, trained *in parallel* on *bootstrap samples* (random samples of size n, drawn *with replacement* from the n training rows)
- **Bagging** = fit each model on its own bootstrap sample
- Each tree is a strong learner; final prediction = *vote* or *average*
- Reduces **variance**

Boosting (XGBoost).

- Many trees, trained *sequentially*; each tree fixes previous errors
- Each tree is a *weak* learner
- Reduces **bias** (and variance with regularization)



Random Forest: many parallel trees averaged. Boosting trains its trees one after another.

Bagging = parallel, reduces variance. Boosting = sequential, reduces bias.

What Is Boosting in Plain English?

The recipe.

1. Start with a **weak learner** (a model that is just-above-chance, slightly better than random – often: predict the mean).
2. Look at what it got **wrong** (the errors).
3. Train a NEW weak learner to **fix** those errors.
4. Add the new model to your prediction.
5. **Repeat** for K iterations.

Key idea. Each new tree only has to worry about the part the previous trees missed.

One sentence: *Many weak learners, applied with discipline, can be extremely strong.*

Why does this work?

A single deep tree is high variance: a slightly different dataset gives a wildly different tree.

A shallow stump (tiny tree) is the opposite: high bias but low variance. Many shallow stumps in series can chase the truth in small steps without overshooting.

Mental picture. Each tree is a tiny correction. The sum of corrections lands on the right answer.

Boosting = take small steps in the direction the model still got wrong.

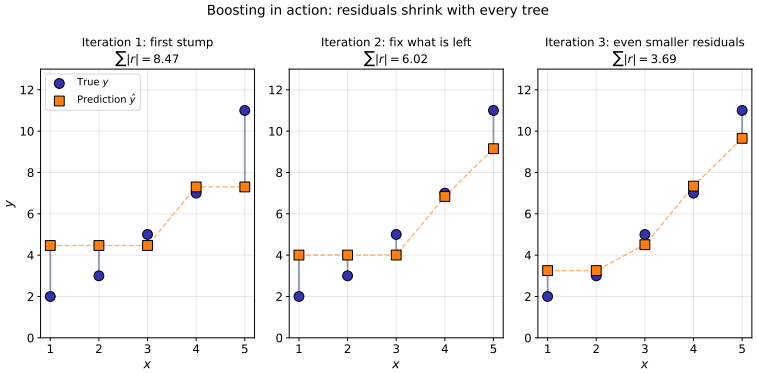
Boosting Intuition Visualized

What the picture shows.

- True values y (purple circles)
- Predictions $\hat{y}$ (orange squares)
- Residuals $r = y - \hat{y}$ (slate vertical bars)

Watch the bars shrink. Each iteration's new tree fits the previous residuals; the bars get shorter.

Same 5-pt dataset we will use in Worked Example A.



Three iterations, one dataset: residuals halve, then halve again.

What Is a Residual?

Definition.

$$r_i = y_i - \hat{y}_i$$

In plain English. The residual is what the current model got *wrong*.

Read the sign.

- $r_i > 0 \Rightarrow$ we predicted *too low* (need to push up)
- $r_i < 0 \Rightarrow$ we predicted *too high* (need to pull down)
- $r_i = 0 \Rightarrow$ correct

Boosting in one line: fit the next tree to the current residuals.

Why fit residuals (not the raw y)?

The residuals are what is *still missing*. The previous trees already explain most of y. Asking the next tree to re-predict y from scratch is wasteful and risks overfitting.

Connection to gradient descent. For squared loss, the negative gradient of the loss with respect to $\hat{y}$ is exactly the residual. In shorthand, $-\partial/\partial\hat{y}_i \frac{1}{2}(y_i - \hat{y}_i)^2 = (y_i - \hat{y}_i) = r_i$:

$$-\frac{\partial}{\partial \hat{y}_i} \frac{1}{2}(y_i - \hat{y}_i)^2 = (y_i - \hat{y}_i) = r_i.$$

So fitting residuals = stepping along the negative gradient.

Residual $r_i = y_i - \hat{y}_i$. The next tree's job: predict r_i.

What Is a Gradient? (1D Warm-up)

Plain English. The **gradient** is just a generalised *slope* (the partial derivative $\partial f/\partial x$). It tells you how steeply the function rises in each direction.

1D warm-up. Take $f(x) = x^2$ at $x = 3$.

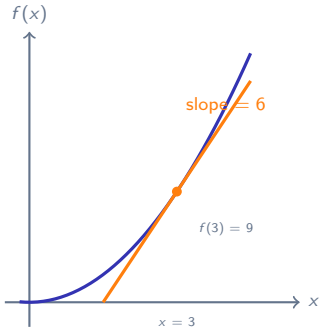
$$f'(x) = 2x \;\Rightarrow\; f'(3) = 6.$$

At $x = 3$, the slope is $+6$ (steeply uphill).

Loss-side reading. For squared loss $\frac{1}{2}(y - \hat{y})^2$:

$$g_i = \frac{\partial l}{\partial \hat{y}_i} = -(y_i - \hat{y}_i) = -r_i.$$

Gradient = −residual for squared loss.



Intuition.

- Big slope ⇒ small step uphill changes f a lot
- Negative gradient direction = downhill
- For loss: stepping with $-g$ reduces loss

Gradient = slope. For squared loss $g = -r$. Stepping $-g$ shrinks loss.

Worked Example A: Setup

Five data points. That is all.

x	1	2	3	4	5
y	2	3	5	7	11

Step 1: initial prediction. Use the mean of y:

$$\hat{y}_i^{(0)} = \bar{y} = \frac{2+3+5+7+11}{5} = \mathbf{5.6}$$

Goal. Fit one stump to the residuals, then update predictions with a learning rate. We will compute every number by hand.

Why start with the mean?

The mean is the constant that minimises squared error. It is the optimal *constant* prediction; everything beyond is a correction.

Why 5.6 exactly?

$$\bar{y} = \frac{28}{5} = 5.6$$

Memorise this number. 5.6 shows up on every following slide.

Five points. Initial prediction = 5.6. Now compute residuals.

Worked Example A Step 1: Residuals

Compute $r_i = y_i - 5.6$ for each point.

x	y	$\hat{y}^{(0)}$	r_i
1	2	5.6	-3.6
2	3	5.6	-2.6
3	5	5.6	-0.6
4	7	5.6	$+1.4$
5	11	5.6	$+5.4$

Residual vector.

$$r = \{-3.6,\ -2.6,\ -0.6,\ +1.4,\ +5.4\}$$

Read the residuals.

- Points 1, 2, 3 are overshot (negative residuals)
- Points 4, 5 are undershot (positive residuals)

Pattern. The residuals grow with x. A split on x should help.

Sanity check. The residuals should sum to zero:

$$-3.6 - 2.6 - 0.6 + 1.4 + 5.4 = 0. \checkmark$$

(They always do when the initial prediction is the mean.)

Residuals tell us which points are still wrong. Now fit a stump to them.

Worked Example A Step 2: Fit a Stump

Candidate split. $x \leq 3$ versus $x > 3$.

Left leaf (points $x = 1, 2, 3$). Mean of residuals:

$$\frac{-3.6 + (-2.6) + (-0.6)}{3} = \frac{-6.8}{3} = \mathbf{-2.27}$$

Right leaf (points $x = 4, 5$). Mean of residuals:

$$\frac{1.4 + 5.4}{2} = \frac{6.8}{2} = \mathbf{+3.40}$$

The first stump.

$$f_1(x) = \begin{cases} -2.27 & x \leq 3 \\ +3.40 & x > 3 \end{cases}$$

Why this split?

For squared loss, the optimal leaf value is the mean of the residuals in that leaf. Splitting at $x \leq 3$ separates negative residuals from positive ones cleanly.

Why a stump? A stump is a tree of depth 1. It can only split once. Boosting works best with shallow trees because each correction is small and controlled.

Sanity check.

- Left leaf prediction is negative $\Rightarrow$ pull the left points DOWN
- Right leaf prediction is positive $\Rightarrow$ push the right points UP

One stump: left leaf -2.27, right leaf $+3.40$. Now apply with a learning rate.

Worked Example A Step 3: The Learning Rate η

Plain English: we don't take the FULL step the stump suggests. We take a *fraction* of it, controlled by a number η (eta).

Definition. η is the **learning rate** (also called *shrinkage*). Typical values: 0.01 to 0.3.

In our example. We use $\eta = 0.5$ (large, for clarity). Real practice: $\eta = 0.1$ is more typical.

What $\eta = 0.5$ means. "Go halfway toward the stump's suggestion."

Why not take the full step?

A single tree is fitted to a single dataset; its proposed correction may be too aggressive. Shrinking the step to $\eta \cdot f_t(x)$:

- Slows down learning – avoids overshooting the truth
- Forces more iterations – gives the ensemble more chances to self-correct
- Acts as a form of regularization

Trade-off. Smaller η means more trees needed for the same fit. Rule of thumb: $\eta \cdot K \approx$ constant.

η = how big a step we take per tree. Smaller η + more trees = safer.

Worked Example A Step 4: Update Predictions

Apply the stump with $\eta = 0.5$.

- Left leaf:

$$\hat{y}^{(1)} = 5.6 + 0.5 \cdot (-2.27) = 5.6 - 1.135 = \mathbf{4.47}$$

- Right leaf:

$$\hat{y}^{(1)} = 5.6 + 0.5 \cdot (+3.40) = 5.6 + 1.70 = \mathbf{7.30}$$

New predictions.

x	y	$\hat{y}^{(1)}$	new r
1	2	4.47	-2.47
2	3	4.47	-1.47
3	5	4.47	$+0.53$
4	7	7.30	-0.30
5	11	7.30	$+3.70$

Did we improve?

Total absolute residual:

- Before:
  $|-3.6| + |-2.6| + |-0.6| + |1.4| + |5.4| = 13.6$
- After:
  $|-2.47| + |-1.47| + |0.53| + |-0.30| + |3.70| = 8.47$

Down by 38%. One stump, with only 50% of its proposed step, already did this much.

Next iteration. Fit *another* stump to the new residuals $\{-2.47, -1.47, +0.53, -0.30, +3.70\}$. Repeat.

One stump, $\eta = 0.5 \Rightarrow$ predictions 4.47 and 7.30. Total residual cut by 38%.

The Residual-Update Formula (Formula F8)

Formula F8.

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(x_i)$$

Decode each piece.

- $\hat{y}_i^{(t)}$ – prediction for point i at iteration t
- $\hat{y}_i^{(t-1)}$ – prediction at the previous iteration
- $f_t(x_i)$ – output of the new tree on point i
- η – learning rate

Read it. New prediction = old prediction + η times the new tree's correction.

F8 in our worked example.

$\eta = 0.5$, f_1 outputs -2.27 on the left, $+3.40$ on the right.

For point $x = 2$ (left):

$$\hat{y}_2^{(1)} = 5.6 + 0.5 \cdot (-2.27) = 4.47.$$

For point $x = 5$ (right):

$$\hat{y}_5^{(1)} = 5.6 + 0.5 \cdot (+3.40) = 7.30.$$

Same formula, different f_t output per leaf.

F8 = the heart of boosting. Update prediction by a fractional step in the new tree's direction.

What F8 Says in Plain English

Strip the symbols away.

F8 says:

"Take the previous prediction. Add a small fraction η of what the new tree predicts. That is your updated prediction."

Three parts.

- Part A: keep the previous prediction
- Part B: build a new tree on the current errors
- Part C: add a SHRUNKEN version of the new tree's output

The shrinkage ($\eta < 1$) is the discipline that keeps the ensemble from overfitting.

Mental model.

Imagine walking down a hill. Each step, you ask: "Which way is downhill from here?" Then you take a half-step (not a full leap) in that direction.

- Old prediction = current position
- New tree = direction of steepest descent
- η = step size

Many small steps are safer than a few giant ones.

F8 = "walk downhill in tiny steps." Each tree is one step.

The Additive Model (Formula F1)

Formula F1.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \qquad f_k \in \mathcal{F}$$

Additive model = SUM of simpler functions. F1 is the canonical **additive model**: $f(x) = f_1(x) + f_2(x) + \ldots + f_K(x)$. Each f_k is a **base learner** (the unit that gets summed – in XGBoost, one decision tree per iteration).

Decode.

- $\hat{y}_i$ – final prediction for point i after all K trees
- f_k – the k-th tree (with the η baked in) – the base learner
- $\mathcal{F}$ – space of regression trees (mapping x to leaf weights)

Read it. Final prediction = SUM of K trees' outputs. Each f_k is a weak learner; summing K of them produces a strong model.

Two crucial properties.

- **Sequential training.** f_1 is fit first; then f_2 given f_1 frozen; then f_3, and so on.
- **Frozen.** Once f_k is trained, it is never re-fit. We only *add* new trees.

This is what differs from Random Forest. RF fits all trees independently and votes; XGBoost fits each tree given the previous predictions.

F1 = additive model. Final $\hat{y}$ is the SUM of K trees, each fitted in turn.

What F1 Says in Plain English

Strip the symbols away.

F1 says:

"Your final answer is the SUM of the answers of all K trees."

Geometry. Each tree carves the feature space into rectangles, and assigns one prediction per rectangle. Adding K trees stacks K such partitions on top of each other.

Effect. The combined boundary is much finer than any single tree could draw.

Reading the picture on the right.

As K grows from 1 to 50:

- $K = 1$ – crude, blocky boundary
- $K = 2$ – a bit smoother
- $K = 5$ – visibly contouring around the orange cluster
- $K = 50$ – smooth, well-fit boundary

Each new tree adds one more layer of correction.

F1 in one sentence: the final prediction is the SUM of all K trees.

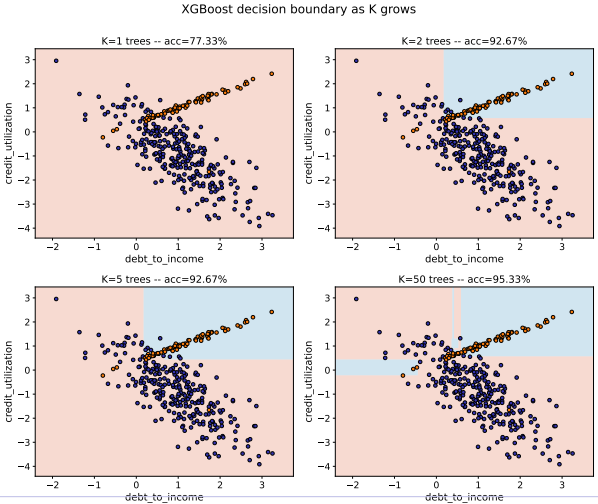
Iteration Progression Visualized

Watch the boundary tighten.

- $K = 1$ – one rectangular split
- $K = 2$ – two stacked corrections
- $K = 5$ – starts to wrap
- $K = 50$ – fully formed

Same data, same hyperparameters.
Only K changes.

No magic. Just F8 applied 50 times.



50 trees, fit one after the other, produce a smooth, accurate boundary.

The XGBoost Objective (Formula F2)

Formula F2.

$$\text{Obj} = \underbrace{\sum_{i=1}^{n} l(y_i, \hat{y}_i)}_{\text{training loss}} + \underbrace{\sum_{k=1}^{K} \Omega(f_k)}_{\text{complexity penalty}}$$

Decode.

- **Loss function** $l(y_i, \hat{y}_i)$ – per-row penalty for how bad the prediction is (squared error, log-loss, ...)
- **Complexity penalty** $\Omega(f_k)$ – tree complexity term that penalises count and magnitude of leaves (defined on next slides)
- **Regularization** = the second term – discourages overfitting by penalty on complexity

Read it. Minimise: training error PLUS a penalty on tree complexity.

This is what distinguishes XGBoost from plain gradient boosting.

Two terms in tension.

- Pure error term: "predict every y perfectly"
- Pure complexity term: "use no trees, all leaves at zero"

The minimum of the sum is a compromise: small enough trees to generalise, deep enough to fit.

Comparison.

- Plain gradient boosting: only the first term
- XGBoost: both terms together (in closed form, see F5–F6)

F2 = error + complexity penalty. XGBoost trades fit against tree size.

Worked Example: F2 on Our 5-Point Dataset

Setup. Use Worked Example A's 5 points after iteration 1 (predictions $\hat{y}^{(1)}$ from earlier).

x	y	$\hat{y}^{(1)}$	$r = y - \hat{y}^{(1)}$
1	2	4.47	-2.47
2	3	4.47	-1.47
3	5	4.47	$+0.53$
4	7	7.30	-0.30
5	11	7.30	$+3.70$

Squared loss per point $l_i = \frac{1}{2} r_i^2$:

$$\tfrac{1}{2}\left(2.47^2 + 1.47^2 + 0.53^2 + 0.30^2 + 3.70^2\right) = \tfrac{1}{2} \cdot 22.5$$

Total training loss.

$$\sum_i l(y_i, \hat{y}_i) \approx \mathbf{11.16}$$

Complexity term. If both stumps had $T = 2$ leaves and weights summed-of-squares ≈ 16:

$$\Omega = \gamma \cdot 2 + \tfrac{1}{2}\lambda \cdot 16$$

With $\gamma = 0$, $\lambda = 1$: $\Omega = 8$.

Total objective F2.

$$\mathrm{Obj} = 11.16 + 8 = \mathbf{19.16}$$

Two competing dials. Loss term wants tiny residuals; complexity term wants tiny weights. F2 sits at the compromise.

F2 numeric: 11.16 **loss** +8 **complexity** = 19.16. **Both dials counted.**

What F2 Says in Plain English

Strip the symbols.

F2 says:

"Minimise the prediction error AND the tree complexity at the same time."

Why penalise complexity?

- Without a penalty, trees can grow until they perfectly memorise noise
- A penalised tree must stop adding leaves when the leaf does not pay for itself
- Smaller trees generalise better

Common analogy. Bias-variance trade-off in one equation.

Two dials, one equation.

- Knob 1: how much loss am I willing to tolerate?
- Knob 2: how much complexity am I willing to pay for?

The penalty parameter (we will call it γ or λ) controls the trade between the two knobs.

Heavier penalty $\Rightarrow$ smaller trees $\Rightarrow$ stronger regularization $\Rightarrow$ less overfitting.

F2 in plain English: balance "fit the data" against "stay simple."

What Is a Gradient? What Is a Hessian?

Plain English.

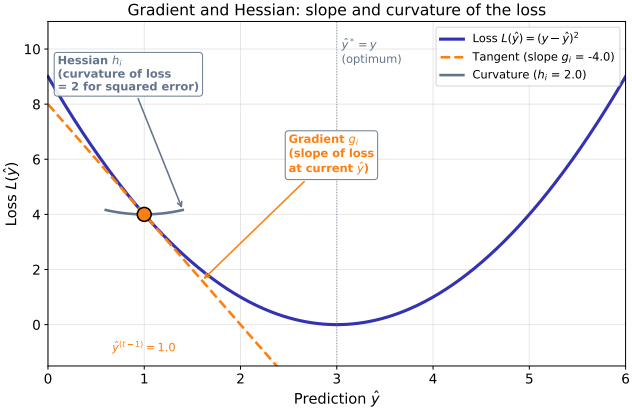
Gradient g_i. The *slope* of the loss with respect to the prediction $\hat{y}_i$ at the current iteration.

Hessian h_i. The *curvature* – how quickly the gradient changes.

Geometry.

- g_i tells us *which direction* reduces loss
- h_i tells us *how confident* we should be in that direction

Plain Newton: step $= -g/h$.



Gradient = slope. Hessian = curvature (the second derivative of loss). Together they make Newton's step.

Second-Order Taylor: 1D Quick Recap

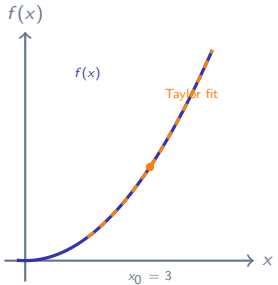
Definition. A **Taylor expansion (2nd order)** is a *local quadratic approximation* of a function $f(x)$ near x_0:

$$f(x_0 + \Delta) \approx f(x_0) + f'(x_0)\,\Delta + \tfrac{1}{2} f''(x_0)\,\Delta^2.$$

Try it. Take $f(x) = x^2$ at $x_0 = 3$, $\Delta = 1$.

- $f(3) = 9$
- $f'(3) = 2 \cdot 3 = 6$
- $f''(3) = 2$
- Quadratic approx: $9 + 6 \cdot 1 + \frac{1}{2} \cdot 2 \cdot 1 = 16$
- Exact: $(3+1)^2 = 16$ ✓

For a quadratic the approximation is exact.



Why care?

- Most losses are NOT quadratic
- But a quadratic FIT around the current prediction is closed-form to optimise
- This is exactly what F3 does for the XGBoost loss

Taylor 2nd order = local quadratic approximation. For $f(x) = x^2$ the fit is exact.

Second-Order Taylor Approximation (Formula F3)

Formula F3.

$$\tilde{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i \, f_t(x_i) + \tfrac{1}{2} h_i \, f_t(x_i)^2 \right] + \Omega(f_t)$$

Where do g_i and h_i come from? In shorthand, $g_i = \partial l / \partial \hat{y}_i$ and $h_i = \partial^2 l / \partial \hat{y}_i^2$ at the previous prediction:

$$g_i = \left. \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i} \right|_{\hat{y}_i^{(t-1)}}$$

$$h_i = \left. \frac{\partial^2 l(y_i, \hat{y}_i)}{\partial \hat{y}_i^2} \right|_{\hat{y}_i^{(t-1)}}$$

First and second derivatives of the per-row loss, evaluated at the previous iteration's prediction.

Why the approximation?

The exact loss after adding tree f_t is hard to optimise directly. Taylor-expanding around the previous prediction gives a quadratic in $f_t(x_i)$, which has a closed-form minimum (see F5).

Key advantage over plain gradient boosting.

- Plain GB: uses g only (first order)
- XGBoost: uses g AND h (second order)
- Result: faster convergence, fewer trees needed

F3 = a quadratic approximation of the loss using slope g and curvature h.

Per-Row Contribution of F3

Where does F3 come from?

Step 1. Apply Taylor 2nd-order to per-row loss at $\hat{y}_i^{(t-1)}$ with $\Delta = f_t(x_i)$. Note: *delta is f_t* (the new tree's output for row i):

$$l(y_i, \hat{y}_i^{(t-1)} + \underbrace{f_t(x_i)}_{\Delta}) \approx l(y_i, \hat{y}_i^{(t-1)}) + \underbrace{g_i}_{\text{slope}} f_t(x_i) + \tfrac{1}{2} \underbrace{h_i}_{\text{curvature}} f_t(x_i)^2.$$

Step 2. Subtract the constant $l(y_i, \hat{y}_i^{(t-1)})$ – it does not depend on f_t, so subtract the constant from the objective.

Step 3. sum over rows gives F3:

$$\tilde{L}^{(t)} \approx \sum_i \left[g_i f_t(x_i) + \tfrac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t).$$

What just happened.

- Started with the exact (untractable) loss $l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i))$
- Replaced it with its 2nd-order Taylor expansion
- Dropped the constant (l at the previous prediction), since it does not affect the minimiser
- What remains is a quadratic in $f_t(x_i)$

Numeric Check (Per-Row Taylor Drop). If $g_i = -0.5$, $h_i = 0.25$, and $f_t(x_i) = 1$:

$$g_i \cdot 1 + \tfrac{1}{2} h_i \cdot 1 = -0.5 + 0.125 = \mathbf{-0.375}.$$

The loss drops by 0.375 on this row.

Drop the constant; keep the quadratic. That's F3.

Per-row Taylor: drop constant, keep quadratic. Sum over rows gives F3.

What F3 Says in Plain English

Strip the symbols.

F3 says:

"Approximate the change in loss caused by adding tree f_t using its slope g AND its curvature h."

Why both?

- Slope alone tells us *which way* to step
- Curvature tells us *how big* a step is safe
- Together: bigger steps when the loss is flat, smaller steps when the loss is curving sharply

Faster convergence than slope alone.

Connection to Newton's method.

A single Newton step on a one-dim quadratic $\frac{1}{2} h\, w^2 + g\, w$ has minimiser:

$$w^* = -\frac{g}{h}$$

This is exactly the leaf-weight formula F5 we will derive next, with the regulariser λ added for stability.

Mental shortcut. Each XGBoost iteration is a (regularised) Newton step in tree space.

F3 in one sentence: use slope AND curvature to step, not just slope.

Regularization Term (Formula F4)

Formula F4.

$$\Omega(f) = \gamma T + \tfrac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Decode.

- T – number of leaves in the tree
- **Leaf weight** w_j – the prediction attached to leaf j (the value output if a point lands in that leaf)
- γ – per-leaf cost; cost per leaf paid for each leaf in the tree (Greek gamma)
- λ – **L2 regularization** (ridge): sum of squared weights $\sum w_j^2$; shrinks weights toward zero (Greek lambda)
- **L1 regularization (Lasso)** (XGBoost `alpha`): penalty proportional to $\sum |w_j|$ – absolute value – can drive weights to exactly zero

Read it. Penalise BOTH the number of leaves AND the magnitude of each leaf's weight.

Two ways a tree can be "too complex."

- Too MANY leaves (over-segmenting the data) – penalised by γT
- Each leaf TOO BIG (huge corrections per region) – penalised by $\lambda \sum w_j^2$

Tuning. Larger γ or $\lambda \Rightarrow$ more conservative trees. Defaults: $\gamma = 0$, $\lambda = 1$ in xgboost.

F4 is what makes XGBoost "X" – eXtreme.

F4 = penalty on leaf count (γ) AND leaf magnitude (λ). Two regularizers in one.

Worked Example: F4 Penalty Calculation

Setup. A tree with $T = 3$ leaves and weights:

$$w = (0.2,\ -0.1,\ 0.3).$$

Hyperparameters $\gamma = 0.5$, $\lambda = 1$.

Per-leaf cost term.

$$\gamma T = 0.5 \cdot 3 = 1.5.$$

L2 penalty term.

$$\sum_j w_j^2 = 0.04 + 0.01 + 0.09 = 0.14.$$

$$\tfrac{1}{2}\lambda \sum_j w_j^2 = \tfrac{1}{2} \cdot 1 \cdot 0.14 = 0.07.$$

Total Ω.

$$\Omega = 1.5 + 0.07 = \mathbf{1.57}.$$

What if we DOUBLE the weights?

New $w = (0.4, -0.2, 0.6)$, same $T = 3$.

$$\sum_j w_j^2 = 0.16 + 0.04 + 0.36 = 0.56.$$

$$\Omega = 1.5 + \tfrac{1}{2} \cdot 0.56 = 1.5 + 0.28 = \mathbf{1.78}.$$

Doubling weights $\Rightarrow$ L2 penalty quadruples ($0.07 \to 0.28$).

What if we add a 4th leaf with $w_4 = 0.0$?

$$\Omega = 0.5 \cdot 4 + 0.07 = 2.07.$$

The γ term punishes the extra leaf even though its weight is zero.

Take-away. Both knobs cost. Choose tree size and weight magnitude carefully.

$\Omega = 1.5 + 0.07 = \mathbf{1.57}$. **Doubling weights** $\to 1.78$. **Adding a leaf** $\to 2.07$.

What F4 Says in Plain English

Strip the symbols.

F4 says:

"Penalise both the number of leaves and the size of each leaf weight."

Two distinct effects.

- γT – discourages adding more leaves (encourages pruning)
- $\frac{1}{2}\lambda \sum w_j^2$ – discourages large leaf predictions (shrinks weights toward zero, just like ridge regression)

Combined effect. Smaller, gentler trees. They cannot "go to extremes" without paying.

Comparison with classical regularization.

- Ridge regression penalises $\sum w_j^2$ ($=$ our λ term)
- Lasso penalises $\sum |w_j|$ (XGBoost has a separate `alpha` parameter for this)
- Decision-tree pruning penalises tree size ($=$ our γ term)

XGBoost combines all three families of regularization in one objective.

F4 in one sentence: shrink BOTH the number of leaves AND the weight per leaf.

Setting Up F5: The Per-Leaf Quadratic

Where do we start? F3 sums per-row contributions. Inside a fixed tree, every row that lands in leaf j gets the SAME prediction w_j.

Step 1: isolate leaf j. For row i in leaf j, replace $f_t(x_i)$ with w_j:

$$\sum_{i \in I_j} \left[g_i w_j + \tfrac{1}{2} h_i w_j^2 \right] + \tfrac{1}{2} \lambda w_j^2.$$

Step 2: factor. w_j doesn't depend on i, so pull it out:

$$L_j(w_j) = \underbrace{G_j}_{\text{total slope}} w_j + \tfrac{1}{2} (\underbrace{H_j + \lambda}_{\text{total curvature}}) w_j^2,$$

with $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$.

L_j is a quadratic in one variable w_j. Easy to minimise.

What just happened.

- Started with a sum over ALL rows (F3)
- Isolated only the rows that land in leaf j
- Used the constant-leaf rule: $f_t(x_i) = w_j$ for every $i \in I_j$
- Factored w_j outside the sum
- Added the λ regulariser

Per-leaf quadratic in w_j. The form $G_j w_j$ plus half $(H_j + \lambda)$ w_j squared is a parabola in w_j – one global minimum.

Next slide. Set $\partial L_j / \partial w_j = 0$ and solve for w_j^* – that's F5.

Per-leaf quadratic: $L_j(w_j) = G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2$. **Now minimise.**

Optimal Leaf Weight (Formula F5)

Formula F5.

$$\boxed{w_j^* = -\frac{G_j}{H_j + \lambda}}$$

where, summing over the points in leaf j:

$$G_j = \sum_{i \in I_j} g_i, \qquad H_j = \sum_{i \in I_j} h_i.$$

Read it. The optimal weight at leaf j is the negative ratio of the leaf's total gradient to its total Hessian (with λ added for stability).

Closed form. No iteration, no gradient descent. Just plug in G_j, H_j, λ.

Where does F5 come from?

For a fixed tree structure, F3 becomes a quadratic in each leaf weight w_j:

$$\sum_{i \in I_j} \left[g_i w_j + \tfrac{1}{2} h_i w_j^2 \right] + \tfrac{1}{2} \lambda w_j^2$$

$$= G_j w_j + \tfrac{1}{2}(H_j + \lambda) w_j^2.$$

Setting *partial L_j over partial w_j equals zero*:

$$\frac{\partial L_j}{\partial w_j} = G_j + (H_j + \lambda) w_j = 0.$$

Now *solve for w_j*:

$$G_j + (H_j + \lambda) w_j = 0 \;\Rightarrow\; w_j^* = -\frac{G_j}{H_j + \lambda}.$$

F5 = closed-form leaf weight. Plug in G_j, H_j, λ. No optimiser needed.

Worked Example: F5 on a Binary-Classif Leaf

Setup. A binary classification leaf with 3 rows. Per-row g_i and h_i from logistic loss with current probabilities p_i:

i	y_i	p_i	$g_i = p_i - y_i$	$h_i = p_i(1-p_i)$
1	0	0.4	$+0.4$	0.24
2	1	0.6	-0.4	0.24
3	1	0.4	-0.6	0.24
Sum			$\Sigma g = -0.6$	$\Sigma h = 0.72$

With $\lambda = 1$:

$$G_j = -0.6, \quad H_j = 0.72.$$

Plug into F5.

$$w_j^* = -\frac{G_j}{H_j + \lambda} = -\frac{-0.6}{0.72 + 1} = \frac{0.6}{1.72}$$

$$= +\mathbf{0.349}.$$

Read it. The leaf wants to push the score UP by 0.349.

- Two of the three rows have $y_i = 1$ (positive class)
- Their probabilities are still too low (under 0.6), so $g_i < 0$
- The leaf weight pushes the prediction toward higher probability

Without λ: $w_j^* = 0.6/0.72 = 0.833$. The λ shrinks the move from 0.83 to 0.35.

Regulariser in action.

$w_j^* = 0.6/(0.72 + 1) = \mathbf{0.349}$. **Regulariser shrinks an unrestrained** $0.833 \to 0.349$.

What Is G_j? What Is H_j?

G_j = **SUM of gradients in leaf j.**

$$G_j = \sum_{i \in I_j} g_i$$

H_j = **SUM of Hessians in leaf j.**

$$H_j = \sum_{i \in I_j} h_i$$

In plain English.

- G_j measures the *total error pressure* pushing the leaf in some direction
- H_j measures the *total certainty* (curvature) of that pressure

λ in the denominator prevents the weight from blowing up when H_j is small (rare leaves).

Squared loss case.

For squared loss with current prediction $\hat{y}_i$:

$$g_i = -(y_i - \hat{y}_i) = -r_i, \qquad h_i = 1.$$

So $G_j = -\sum_{i \in I_j} r_i$ and $H_j = |I_j|$ (number of points in the leaf).

With $\lambda = 0$:

$$w_j^* = \frac{\sum_{i \in I_j} r_i}{|I_j|} = \bar{r}_j$$

the **average residual** in the leaf – exactly the rule we used in Worked Example A.

G_j and H_j are just the totals over points in leaf j. F5 uses both.

Numeric Check: F5 Recovers Worked Ex A's Stump

Setup. Squared loss case. Per-row gradient and Hessian:

$$g_i = -(y_i - \hat{y}_i) = -r_i, \qquad h_i = 1, \qquad \lambda = 0.$$

Left leaf (rows $i = 1, 2, 3$), residuals $r = (-3.6, -2.6, -0.6)$:

$$G_L = -\sum r_i = 3.6 + 2.6 + 0.6 = 6.8.$$

$$H_L = 3, \quad w_L^* = -\frac{G_L}{H_L} = -\frac{6.8}{3} = -\mathbf{2.27}.$$

Right leaf (rows $i = 4, 5$), residuals $r = (1.4, 5.4)$:

$$G_R = -(1.4 + 5.4) = -6.8, \quad H_R = 2.$$

$$w_R^* = -\frac{-6.8}{2} = +\mathbf{3.40}.$$

Compare.

Worked Example A computed leaf weights from the *average residual* rule:

- Left: mean of $\{-3.6, -2.6, -0.6\}$ $= -6.8/3 = -2.27$
- Right: mean of $\{1.4, 5.4\} = 6.8/2 = +3.40$

F5 with squared loss ($g = -r$, $h = 1$, $\lambda = 0$) gives:

$$w_j^* = -\frac{-\sum r_i}{|I_j|} = \frac{\sum r_i}{|I_j|} = \bar{r}_j.$$

-2.27 and $+3.40$ – exact match.

F5 is the general formula; the average-residual rule is the special case for squared loss with no regulariser.

F5 with $g = -r$, $h = 1$, $\lambda = 0$: $w_L^* = -2.27$, $w_R^* = +3.40$ – matches Worked Ex A.

Derivation of F6

Step 1. *Plug w_j star into L_j.* Recall $w_j^* = -G_j/(H_j + \lambda)$. Substitute back:

$$L_j(w_j^*) = G_j w_j^* + \tfrac{1}{2}(H_j + \lambda)(w_j^*)^2.$$

Step 2. *Combine the two G_j squared terms.* Plugging in:

$$= G_j \cdot \frac{-G_j}{H_j + \lambda} + \tfrac{1}{2}(H_j + \lambda)\frac{G_j^2}{(H_j + \lambda)^2}$$

$$= -\frac{G_j^2}{H_j + \lambda} + \frac{1}{2}\frac{G_j^2}{H_j + \lambda} = -\tfrac{1}{2}\frac{G_j^2}{H_j + \lambda}.$$

Step 3. *Sum over leaves* adds γT:

$$\mathrm{Obj}^* = -\tfrac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T.$$

What just happened.

- L_j at $w_j = w_j^*$ is the minimum value the per-leaf quadratic can take
- Two $G_j^2/(H_j + \lambda)$ terms appear, with opposite signs and a $\frac{1}{2}$
- They combine to a single $-\frac{1}{2}\,G_j^2/(H_j + \lambda)$
- Summing over all leaves $j = 1 \ldots T$, plus the γT from Ω, gives F6

F6 says: the lowest objective achievable for a fixed tree shape depends only on the per-leaf totals G_j, H_j and the regularisers.

Why minus? Lower (more negative) Obj* = better tree. The $-\frac{1}{2}\,G_j^2$ term is what drives the gain.

Plug w_j^* in, two G_j^2 terms combine, sum over leaves $\Rightarrow$ F6.

Optimal Objective Value (Formula F6)

Formula F6. Plug w_j^* from F5 back into F3:

$$\text{Obj}^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \gamma T$$

Read it. The minimum achievable objective for a fixed tree structure depends only on the per-leaf totals G_j, H_j, the regulariser λ, and the tree size T.

Lower Obj^* = better tree structure.

This number is what we compare across candidate splits.

How XGBoost uses F6.

For every candidate split:

1. Compute G and H for the parent leaf
2. Compute G_L, H_L, G_R, H_R for the candidate left and right children
3. Compute the change in Obj^* when the parent is replaced by L + R

That change is the **Gain** (formula F7 on the next slide). If Gain > 0, take the split.

Greedy. XGBoost scans every feature $\times$ every threshold and picks the split with the largest Gain.

F6 = the minimum loss for a given tree structure. Smaller Obj^* = better tree.

Worked Example: F6 on Our 2-Leaf Tree

Setup. The Worked Example A stump has $T = 2$ leaves with totals (squared loss, $\lambda = 0$):

- Left: $G_L = 6.8$, $H_L = 3$
- Right: $G_R = -6.8$, $H_R = 2$

For the Numeric Check use $\lambda = 0$, and let $\gamma = 0.5$ to see the leaf cost.

Per-leaf $G_j^2/(H_j + \lambda)$.

$$\text{Left: } \frac{6.8^2}{3} = \frac{46.24}{3} = 15.41$$

$$\text{Right: } \frac{(-6.8)^2}{2} = \frac{46.24}{2} = 23.12$$

Sum and apply F6.

$$\sum_j \frac{G_j^2}{H_j + \lambda} = 15.41 + 23.12 = 38.53.$$

$$\text{Obj}^* = -\tfrac{1}{2} \cdot 38.53 + \gamma \cdot 2.$$

With $\gamma = 0.5$:

$$\text{Obj}^* = -19.265 + 1.0 = \mathbf{-18.265}.$$

Sanity.

- Negative Obj* = the tree REDUCES loss
- Bigger negative = better
- The γT term penalises but doesn't dominate

F6 = how good a tree structure is, in one number.

Obj* $= -\frac{1}{2}(15.41 + 23.12) + 0.5 \cdot 2 = -19.265 + 1.0 = \mathbf{-18.265}.$

Deriving F7 from F6

Idea. Gain = how much Obj* DROPS when we replace one parent leaf with two children L, R.

Step 1. *subtract Obj star of parent* from Obj star with L+R children:

- Parent (1 leaf): $-\frac{1}{2} \cdot \frac{(G_L+G_R)^2}{H_L+H_R+\lambda} + \gamma \cdot 1$
- Children (2 leaves): $-\frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda}\right] + \gamma \cdot 2$

Step 2. Compute the difference (parent − children):

$$-\frac{1}{2}\left[\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \left(-\frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda}\right]\right) + \gamma(1-2).$$

Step 3. Simplify. The γ cancels except one leaf cost ($-\gamma$ remains for the extra leaf). Result:

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma.$$

What just happened.

- Wrote Obj* before the split (1 parent leaf)
- Wrote Obj* after the split (2 children L, R)
- Subtracted: parent minus children = the DROP in Obj*
- Two γT terms: $\gamma \cdot 1$ (parent) vs $\gamma \cdot 2$ (children); difference is $-\gamma$

Result: F7.

- Quality of left + quality of right − quality of parent
- Halved (because $\frac{1}{2}$ in F6)
- Minus γ (cost of one new leaf)

Decision rule. Gain $> 0 \Rightarrow$ split worth it.

F7 = parent−children, $\frac{1}{2}$ scaled, minus γ for one extra leaf.

The Split Gain Formula (Formula F7)

Formula F7. Splitting one parent leaf into Left (L) and Right (R) reduces Obj^* by:

$$\mathrm{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

Decode the three fractions.

- $G_L^2/(H_L+\lambda)$ – "quality" of the left child
- $G_R^2/(H_R+\lambda)$ – "quality" of the right child
- $(G_L+G_R)^2/(H_L+H_R+\lambda)$ – "quality" of the parent (no split)

Subtract γ. The cost of adding a new leaf.

Decision rule.

- Gain $> 0 \Rightarrow$ take the split (it improves the objective)
- Gain $\leq 0 \Rightarrow$ do NOT split (it does not pay for itself)

Why γ matters. If γ is large, only *very informative* splits will survive. This is XGBoost's built-in pruning.

Pre-pruning by formula. No need for post-hoc tree pruning – a split that doesn't pay just doesn't happen.

F7 = (left + right) - parent quality, minus the leaf cost γ. Positive Gain = take it.

What F7 Says in Plain English

Strip the symbols.

F7 says:

"Gain = quality of LEFT child + quality of RIGHT child − quality of PARENT − cost of a new leaf."

Decision.

- If *positive* ⇒ make the split
- If *negative* ⇒ don't

Brute-force search.

- Try every feature
- Try every threshold (between adjacent sorted values)
- Compute Gain
- Pick the split with the LARGEST Gain

Three building blocks per leaf.

The "quality" $G^2/(H+\lambda)$ is just F6 evaluated for one leaf. It says:

"How much would adding THIS leaf reduce the loss?"

A split is worth it when the two children together explain more than the parent alone, by at least γ.

One picture. Imagine you have a slot. The split asks: "If I cut this slot in two, can I fit more total signal in the two halves than I could in the original slot?"

F7 in plain English: take a split only if children + children − parent beats γ.

Worked Example B: Compute Gain for One Split

Setup. A candidate split with the following per-child totals:

$$G_L = 3.0, \quad H_L = 4.0$$

$$G_R = -2.0, \quad H_R = 3.0$$

Regularizers.

$$\lambda = 1.0, \qquad \gamma = 0.5$$

Goal. Compute the Gain. Decide: take the split or not?

Plan. Compute three terms (left, right, parent), combine, subtract γ.

Why these numbers?

For a binary classification task with log-loss, g_i and h_i depend on the previous prediction:

$$g_i = p_i - y_i, \qquad h_i = p_i(1 - p_i)$$

where $p_i = \sigma(\hat{y}_i^{(t-1)})$.

The numbers shown (e.g., $G_L = 3.0$) are realistic-scale aggregates over a leaf's data after one or two iterations.

The arithmetic that follows is what XGBoost does on every split it scans.

Inputs: $G_L = 3$, $H_L = 4$, $G_R = -2$, $H_R = 3$, $\lambda = 1$, $\gamma = 0.5$. **Compute Gain.**

Worked Example B Step 1: Left Child Quality

Formula.

$$\frac{G_L^2}{H_L + \lambda}$$

Plug in. $G_L = 3.0$, $H_L = 4.0$, $\lambda = 1.0$.

$$\frac{(3.0)^2}{4.0 + 1.0} = \frac{9.0}{5.0} = \mathbf{1.80}$$

Interpretation. The left child can reduce the objective by an amount proportional to 1.80. The bigger this number, the more useful the left leaf is.

Sanity checks.

- Numerator $G_L^2 = 9$ is non-negative (square)
- Denominator $H_L + \lambda = 5$ is positive
- Result $1.80 > 0$

Effect of λ. If λ were larger (say $\lambda = 5$), the denominator becomes 9 and the quality drops to $9/9 = 1.0$. Stronger regularization $\Rightarrow$ smaller leaf quality.

Carry 1.80 into the Gain formula.

Left child quality $= 9/5 = 1.80$. One of three terms in F7.

Worked Example B Step 2: Right Child Quality

Same formula, different leaf.

$$\frac{G_R^2}{H_R + \lambda}$$

Plug in. $G_R = -2.0$, $H_R = 3.0$, $\lambda = 1.0$.

$$\frac{(-2.0)^2}{3.0 + 1.0} = \frac{4.0}{4.0} = \mathbf{1.00}$$

Note the squaring. The sign of G_R does NOT matter for the quality term; only $|G_R|$ does. Both leaves contribute non-negative quality.

Why the sign of G disappears.

The optimal leaf weight $w^* = -G/(H + \lambda)$ uses the sign. But the *quality* term $G^2/(H + \lambda)$ measures how much the loss drops, which only cares about the magnitude.

Both directions are fine.

- Negative G: leaf wants positive weight
- Positive G: leaf wants negative weight

Both reduce loss equally if magnitudes match.

Carry 1.00 into the Gain formula.

Right child quality $= 4/4 = 1.00$. Two of three terms collected.

Worked Example B Step 3: Parent Quality (No Split)

Formula.

$$\frac{(G_L + G_R)^2}{H_L + H_R + \lambda}$$

Plug in.

$$G_L + G_R = 3.0 + (-2.0) = 1.0$$

$$H_L + H_R + \lambda = 4.0 + 3.0 + 1.0 = 8.0$$

$$\frac{(1.0)^2}{8.0} = \frac{1.0}{8.0} = \mathbf{0.125}$$

Interpretation. Without splitting, this leaf's quality is only 0.125. The two children can do much better than that.

Why is the parent quality so low?

$G_L = +3.0$ and $G_R = -2.0$ point in OPPOSITE directions. When we sum them, they cancel: $G_L + G_R = 1.0$.

The parent leaf can only set ONE weight w for both halves, so the conflicting signals dilute each other.

This is why splitting helps. The split lets each half have its own weight, recovering the magnitude lost to cancellation.

Three numbers ready: 1.80, 1.00, 0.125.

Parent quality $= 1/8 =$ 0.125. Three terms ready, plug into Gain.

Worked Example B Step 4: Compute Gain

Plug into F7.

$$\text{Gain} = \tfrac{1}{2}(1.80 + 1.00 - 0.125) - 0.5$$

Inside the bracket.

$$1.80 + 1.00 - 0.125 = 2.675$$

Halve it.

$$\tfrac{1}{2} \cdot 2.675 = 1.3375$$

Subtract $\gamma = 0.5$.

$$1.3375 - 0.5 = \mathbf{0.8375}$$

Gain $= 0.8375 > 0 \Rightarrow$ KEEP the split.

What just happened.

Splitting the parent into L + R raises the total leaf quality from 0.125 (cancelled gradients) to $1.80 + 1.00 = 2.80$ (each half gets its own weight).

The improvement is $2.80 - 0.125 = 2.675$, halved (because of the $\frac{1}{2}$ in F6) to 1.3375, minus the γ cost of one extra leaf, leaves 0.8375.

Decision. Positive Gain $\Rightarrow$ XGBoost would take this split.

Try yourself. Set $\gamma = 1.5$ instead. What happens? (Gain becomes -0.16 – the split is rejected.)

$\text{Gain} = 0.5 \cdot 2.675 - 0.5 = \mathbf{0.8375}$. **Positive $\Rightarrow$ split kept.**

Numeric Check: F7 With Larger γ (Split Rejected)

Same numbers, bigger γ. Re-use Worked Example B totals: $G_L^2/(H_L+\lambda) = 1.80$, $G_R^2/(H_R+\lambda) = 1.00$, parent term $= 0.125$. We showed Gain $= 0.8375$ when $\gamma = 0.5$.

Now set $\gamma = 1.5$. This means *each new leaf costs 1.5 objective units* – tripled from before.

$$\text{Gain} = \tfrac{1}{2}(1.80 + 1.00 - 0.125) - 1.5$$

$$= \tfrac{1}{2} \cdot 2.675 - 1.5 = 1.3375 - 1.5 = \mathbf{-0.1625}.$$

Gain $= -0.1625 < 0 \Rightarrow$ REJECT the split.

Pre-pruning in action.

Nothing about the *data* changed. Only γ changed.

- With $\gamma = 0.5$: split kept (Gain $= +0.8375$)
- With $\gamma = 1.5$: split rejected (Gain $= -0.1625$)

The γ dial. Larger γ = stricter standard for allowing new leaves. Splits that barely improve the objective stop happening.

This is why we call it pre-pruning. The tree grows only as deep as the splits can pay for their own leaf cost.

Raise γ to regularise structure; lower it to grow bushier trees.

Same data, γ: 0.5 $\rightarrow$ 1.5 flips keep-split into reject-split. Pre-pruning by formula.

Logistic-Loss Gradients

Goal. Derive the logistic (cross-entropy) g_i and h_i used when XGBoost does binary classification.

Logistic loss. For margin $\hat{y}_i$ and $p_i = \sigma(\hat{y}_i) = 1/(1+e^{-\hat{y}_i})$:

$$l(\hat{y}_i) = -\underbrace{[y_i \log p_i + (1-y_i)\log(1-p_i)]}_{\text{binary cross-entropy}}.$$

Chain rule through sigmoid. Use $\partial p_i/\partial \hat{y}_i = p_i(1-p_i)$:

$$\frac{\partial l}{\partial \hat{y}_i} = -\left[\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i}\right] \underbrace{p_i(1-p_i)}_{\partial p/\partial \hat{y}}.$$

Simplify the bracket: $y_i(1-p_i) - (1-y_i)p_i = y_i - p_i$. So

$$g_i = p_i - y_i, \qquad h_i = p_i(1-p_i).$$

g equals p minus y and **h equals p times one minus p**.

Numeric Check. Take $\hat{y}_i = 0$ so $p_i = \sigma(0) = 0.5$. Suppose the true label is $y_i = 1$:

$$g_i = 0.5 - 1 = -\mathbf{0.5}, \quad h_i = 0.5 \cdot 0.5 = \mathbf{0.25}.$$

What each number says.

- $g_i < 0 \Rightarrow$ push $\hat{y}_i$ UP (since $y_i = 1$)
- $h_i = 0.25 =$ max curvature of logistic loss (at $p = 0.5$)

Boundary cases.

- $p_i = 0.01$, $y_i = 0$: $g = +0.01$ (almost right, tiny push)
- $p_i = 0.99$, $y_i = 0$: $g = +0.99$ (very wrong, big push)

$g = p - y$ **is a signed miss.** Zero when $p = y$, large when far off.

$g_i = p_i - y_i$, $h_i = p_i(1-p_i)$. **Plug into F5/F6/F7 for classification.**

Numeric Check: Sigmoid Value Table

What is σ? The *sigmoid* squashes any real number z into a probability in $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Five reference values.

z	e^{-z}	$\sigma(z) = 1/(1 + e^{-z})$
-2	7.389	**0.12**
-1	2.718	**0.27**
0	1.000	**0.50**
$+1$	0.368	**0.73**
$+2$	0.135	**0.88**

Symmetry: $\sigma(-z) = 1 - \sigma(z)$.

Read the table.

- $z = 0 \Rightarrow p = 0.5$ (maximum uncertainty)
- $|z|$ large positive $\Rightarrow p$ close to 1
- $|z|$ large negative $\Rightarrow p$ close to 0

How XGBoost uses this. After K trees the *margin* is $\hat{y}_i = \sum_k \eta f_k(x_i)$, a real number. To get a probability:

$$p_i = \sigma(\hat{y}_i).$$

Worked example. Five trees with $\eta = 0.1$ vote $+4, +3, +2, -1, +4$. Margin $\hat{y} = 0.1 \cdot 12 = 1.2$. So

$$p = \sigma(1.2) \approx 0.77.$$

Margin $\rightarrow$ probability via one sigmoid at the end.

σ: $-2 \rightarrow 0.12$, $0 \rightarrow 0.5$, $+2 \rightarrow 0.88$. **Final-layer squash.**

XGBoost on Our Credit Dataset

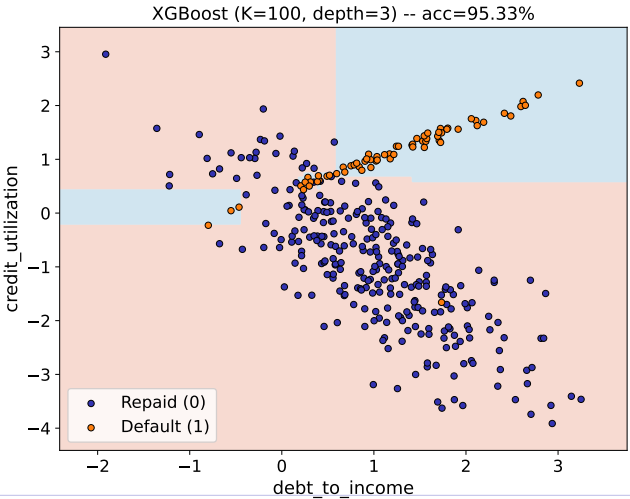
What you see.

- Smooth boundary (smoother than a single tree)
- Slightly different shape from Random Forest – boosting carves more carefully
- 100 trees, depth 3, $\eta = 0.1$

Same numbers everywhere:

- Tree, RF, XGBoost all hit $\approx 95\%$ on this clean dataset
- Differences emerge on harder problems

XGBoost edges out RF on noisier or higher-dim data.



Smoother than a single tree. Each iteration adds another layer of refinement.

Training and Validation Loss: When to Stop

Two curves to read.

- **Training loss (purple)** – decreases monotonically as trees pile up
- **Validation loss (slate)** – U-shaped: improves, then worsens

Early stopping.

- Monitor validation loss after each tree
- Stop after `early_stopping_rounds` iterations with no improvement

Orange dashed line = the iteration where validation loss is minimised.

Train monotone, val U-shape. Early stopping picks the U-shape minimum automatically.

Hyperparameters Part 1

Parameter	Effect	Bias-Variance
`n_estimators` (K)	More trees: fits training better, slower	Bias ↓, Var ↑
`learning_rate` (η)	Smaller η: smoother, needs more trees	Var ↓, Bias ↑
`max_depth`	Deeper trees: more interactions captured	Bias ↓, Var ↑
`min_child_weight`	Min Hessian per leaf: larger = conservative	Var ↓, Bias ↑

Key paired controls.

- K and η usually move TOGETHER: small η + many trees + early stopping is the safest default
- `max_depth` controls how complex EACH tree can be (3–8 typical)
- `min_child_weight` prevents leaves from being created on $<$ some threshold of evidence

Default starting recipe: $K = 1000$, $\eta = 0.05$, `max_depth` $= 5$, with `early_stopping_rounds` $= 50$ on a held-out set.

Pairs of dials. Tune η and K together; tune `max_depth` once.

Hyperparameters Part 2

Parameter	Effect	Bias-Variance
`gamma` (γ)	Min Gain to split: larger = more pruning	Var ↓, Bias ↑
`lambda` (λ)	L2 penalty on leaf weights	Var ↓, Bias ↑
`subsample`	Fraction of rows per tree (bagging-like)	Var ↓
`colsample_bytree`	Fraction of columns per tree	Var ↓

Three families of regularization.

- **Pruning regularization.** γ is the per-leaf cost in F7. Larger γ = fewer leaves.
- **Weight regularization.** λ shrinks leaf weights toward zero.
- **Stochastic regularization.** `subsample` and `colsample_bytree` add bagging-style noise to each tree – a built-in form of dropout for trees.

Common defaults: $\gamma = 0$, $\lambda = 1$, `subsample` $= 0.8$, `colsample_bytree` $= 0.8$. Tune with grid search or Bayesian optimisation.

Three flavours of regularization. Subsample + colsample act like bagging on top of boosting.

Learning Rate Sweep

What you see.

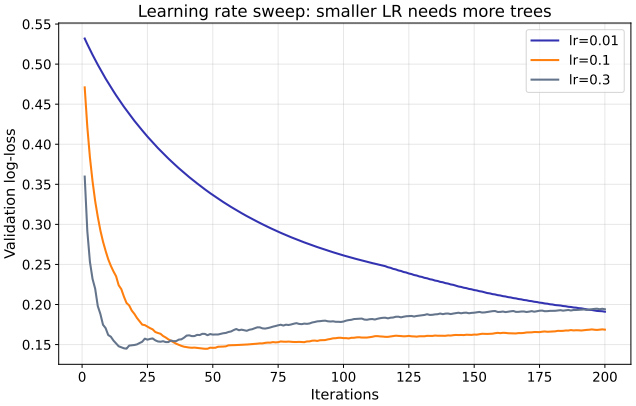
- $\eta = 0.01$ – slow learner, needs many trees
- $\eta = 0.10$ – moderate, hits min around $K = 50$
- $\eta = 0.30$ – fast learner, hits min early then plateaus

Pattern.

$$\eta \cdot K \approx \text{constant}$$

for the optimal stopping point.

Rule of thumb: smaller η + more trees with early stopping is the safer default.



Smaller learning rate ⇒ more trees needed. Use early stopping to find the sweet spot.

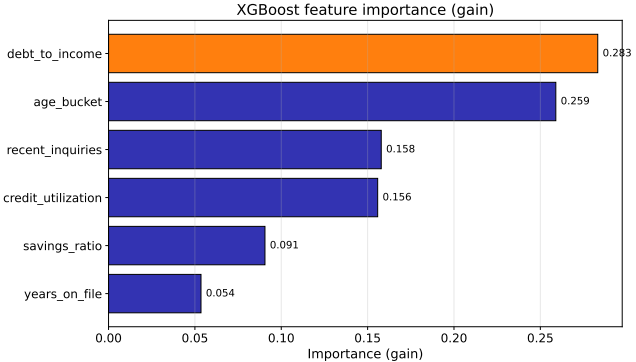
Feature Importance and SHAP

Three flavours of importance.

- **weight** – how often a feature is used in a split
- **gain** – average objective improvement from splits on that feature (default in xgboost)
- **cover** – average Hessian weight on splits for that feature

SHAP. Per-prediction attribution: for each row, each feature gets a value telling how much it pushed the prediction up or down. Global SHAP bar = mean $|\mathrm{SHAP}|$ across rows.

Use SHAP for "why this borrower?" Use gain importance for "which features matter on average."



Gain = global default. SHAP = per-row explanation. Different questions, different tools.

Worked Example: SHAP Row Attribution

One borrower, one prediction. A single row from the credit dataset. XGBoost margin $\hat{y} = 0.5$, so probability $p = \sigma(0.5) \approx 0.62$.

SHAP decomposition. SHAP splits $\hat{y}$ into a baseline (average margin over the training set) plus per-feature contributions that SUM to the margin.

$$\hat{y} = \underbrace{\phi_0}_{\text{baseline}} + \sum_{k=1}^{F} \phi_k.$$

This row. Baseline $\phi_0 = 0$. Three features:

- *debt_ratio*: $\phi_{\text{debt}} = +0.6$ (raises default risk)
- *credit_history*: $\phi_{\text{hist}} = -0.3$ (good history lowers risk)
- *savings*: $\phi_{\text{sav}} = +0.2$ (borderline)

$$\hat{y} = 0 + 0.6 - 0.3 + 0.2 = \mathbf{0.5}, \quad p = \sigma(0.5) \approx \mathbf{0.62}.$$

Read the attribution.

- Positive $\phi \Rightarrow$ feature pushes prediction UP
- Negative $\phi \Rightarrow$ feature pushes prediction DOWN
- $|\phi| \Rightarrow$ magnitude of the push

Sanity check. The three ϕ_k sum to 0.5, matching the model's margin for this row. *This is the SHAP axiom:* contributions sum to the prediction.

Comparison to global importance.

- *Gain importance* (global): debt_ratio might rank first because it splits the tree best OVERALL.
- *SHAP* (this row): debt_ratio still dominant, but credit_history PARTIALLY cancels it. Per-row story.

Same model, different question, different tool.

Baseline + 0.6 - 0.3 + 0.2 = 0.5 $\Rightarrow$ p = σ(0.5) $\approx$ 0.62. SHAP sums to margin.

5-Method Comparison on Our Dataset

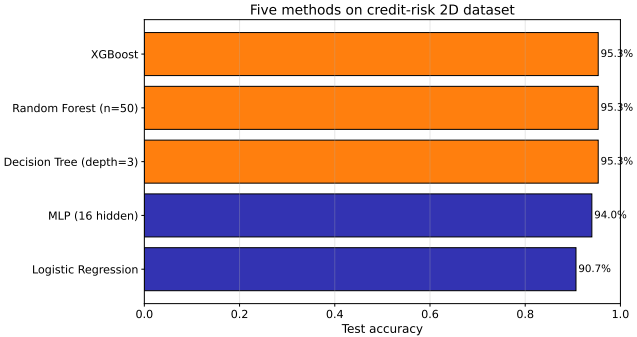
What is plotted.

Test accuracy of five models on the 2D credit dataset:

- **Logistic Regression** – linear floor
- **Decision Tree (depth 3)** – single tree
- **Random Forest (50)** – bagged trees
- **MLP (16 hidden)** – shallow neural net
- **XGBoost** (orange winner)

Take-away. On clean tabular data, tree ensembles tie. On noisier or higher-dim problems, XGBoost typically pulls ahead.

XGBoost first; switch to NNs only for images, text, or audio.



Tabular ⇒ XGBoost first. Images, text, audio ⇒ neural nets.

When to Use XGBoost (and When NOT)

Use XGBoost when:

- Tabular data, small-to-medium size (100s to millions of rows)
- Mixed feature types (numeric + categorical)
- Some missing values
- Need strong baseline fast
- Interpretability via gain or SHAP is acceptable

Typical wins. Credit scoring, fraud detection, churn prediction, sales forecasting on tabular features.

Avoid XGBoost when:

- Images, audio, raw text $\Rightarrow$ CNNs / transformers win
- Truly linear relationships $\Rightarrow$ logistic / OLS faster + interpretable
- Very small data (< 100 rows) $\Rightarrow$ simpler models avoid overfitting
- Sub-millisecond latency $\Rightarrow$ compiled rules or tiny linear models
- $n > 10^7$ rows $\Rightarrow$ training time + memory become prohibitive

Right tool for the job. XGBoost is the default first try on tabular – but not the only hammer.

Tabular wins. Images, text, very-small or very-large data: pick something else.

Summary: Key Takeaways

Five things to remember.

1. **Boosting** = sequential trees, each fitting the residuals of the previous ensemble.
2. **F1 (additive model)** – final prediction is the SUM of all K trees.
3. **F2 (objective)** – minimise loss PLUS complexity penalty Ω.
4. **F3 (Taylor)** – approximate the loss using both gradient g AND Hessian h.
5. **F7 (Gain)** – a split is taken only if children $-$ parent $> \gamma$.

Three numbers that anchor everything.

- Worked Example A initial: $\bar{y} = \mathbf{5.6}$
- Worked Example A after one stump: predictions **4.47** and **7.30**
- Worked Example B Gain: **0.8375**

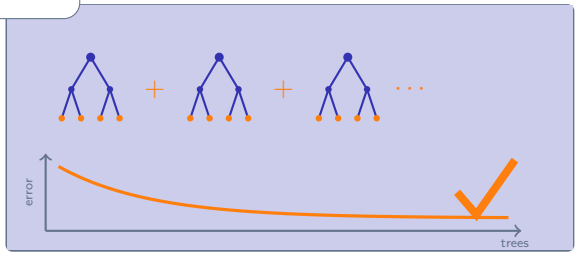
Use it. On your next tabular problem:

- Start with $K = 1000$, $\eta = 0.05$, depth 5
- Use `early_stopping_rounds` $= 50$
- Tune γ, λ if validation says so

Weak learners + discipline = strong ensemble.

Eight formulas, two worked examples, one rule: tabular → XGBoost first.

Weak learners + discipline = strong ensemble.



Each tree is a small correction; their SUM is a powerful classifier.

What Is an Ensemble?

Definition. An *ensemble* is a collection of many simple models whose outputs are combined (sum, average, or vote) into a single prediction. No single model dominates; the *combination* is the answer.

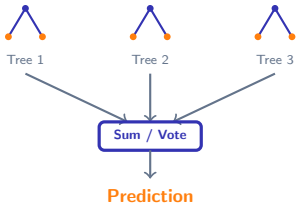
Why combine? A single weak learner might be 55% accurate. But *three* independent 55%-accurate stumps that VOTE push accuracy toward $\sim 57\%$ and up – each captures different errors.

Two flavours.

- *Bagging* (Random Forest): trees vote independently.
- *Boosting* (XGBoost): trees are *sequential*, each fixing the previous ensemble's mistakes.

"Ensemble" = simple models cooperating. Combined prediction beats any one member.

Picture.



Numeric. 3 stumps, each independently right 55% of the time, voting MAJORITY: correct when ≥ 2 agree. Probability ≈ 0.57.

XGBoost = a very disciplined boosting ensemble.

Ensemble = combine many weak models. Bagging (parallel) vs boosting (sequential).

L1 vs L2 vs γ: Three Flavours of Regularization

Three penalties. One goal: simpler trees.

L2 (ridge)	L1 (Lasso)	γ (leaf cost)
$\lambda \sum_j w_j^2$	$\alpha \sum_j \lvert w_j \rvert$	γT
Squared weight Shrinks toward 0 XGB: `reg_lambda`	Abs weight Pushes some to 0 XGB: `reg_alpha`	Per-leaf flat fee Prunes entire leaves XGB: `gamma`

Worked numeric. Three leaves, weights $w = (0.2, -0.1, 0.3)$, $T = 3$.

- $L2 = \lambda(0.04 + 0.01 + 0.09) = 0.14\lambda$
- $L1 = \alpha(0.2 + 0.1 + 0.3) = 0.6\alpha$
- γ-term $= 3\gamma$

L2 shrinks; L1 sparsifies; γ prunes.

When does each dominate?

- **L2 (λ)**: dampens all weights smoothly. Default regulariser. Good when features are correlated.
- **L1 (α)**: forces small weights to exactly 0. Useful for feature selection at the leaf level.
- γ: multiplies the *number of leaves*. Raise to get shallower, pre-pruned trees.

Practical recipe.

- Start with $\lambda = 1$, $\alpha = 0$, $\gamma = 0$
- Overfitting? Raise γ (structural pruning) and λ (weight shrinkage)
- Rarely need all three simultaneously

Three knobs, same goal: keep the ensemble honest.

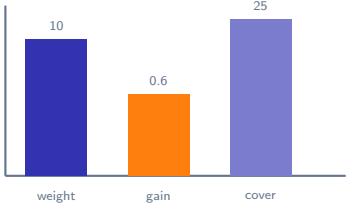
L2 = λw^2; L1 = $\alpha|w|$; γT = flat leaf cost. Tune one at a time.

Feature Importance: Three Flavours Explained

XGBoost reports three importance metrics. They answer different questions.

- *weight* – number of splits that use the feature (frequency)
- *gain* – average Obj* drop per split on the feature (impact)
- *cover* – average Hessian mass on splits using the feature (reach)

Same feature, three numbers. For one feature X:



Read the bars.

- Feature X was used in **10** splits (weight)
- Average Obj* drop per split: **0.6** (gain)
- Average Hessian on those splits: **25** (cover)

Which to trust? *gain* is the default: it captures impact, not frequency.

Pitfall. A feature used in many splits (high weight) but with tiny gain is just being *greedy-matched* on noise. Trust gain.

And SHAP? SHAP is the per-row story (see frame 59). *gain* is the global story.

Importance = which feature helps the objective most, not which is used most.

weight = count, gain = avg Obj drop, cover = avg Hessian. Default to gain.

Anatomy of an XGBoost Prediction

A prediction in five steps.

1. Input row x.
2. Each of K trees outputs a leaf value $f_k(x)$.
3. Multiply by η (shrinkage): $\eta f_k(x)$.
4. SUM the K contributions $\Rightarrow$ *margin* $\hat{y}$.
5. Apply σ: *probability* $p = \sigma(\hat{y})$.

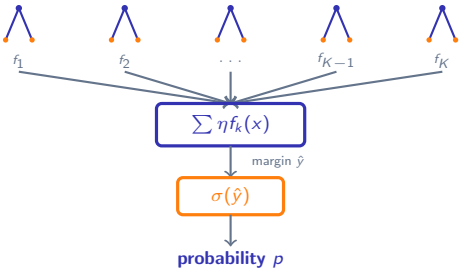
Numeric walk-through. Five trees, $\eta = 0.1$, leaf outputs $(+4, +3, +2, -1, +4)$:

$$\hat{y} = 0.1 \cdot (4 + 3 + 2 - 1 + 4) = 0.1 \cdot 12 = \mathbf{1.2}.$$

$$p = \sigma(1.2) \approx \mathbf{0.77}.$$

Sum the trees, squash with sigmoid.

Picture.



Trees $\rightarrow$ η-scaled $\rightarrow$ sum (margin) $\rightarrow$ sigmoid (probability). One pipeline.

Why 'Extreme' Gradient Boosting? Key Speedups

What makes XGBoost extreme? Four engineering tricks on top of plain gradient boosting.

1. **Parallel split finding.** Within one tree, candidate splits for different features are scanned in parallel across CPU cores.
2. **Cache-aware access.** Data is laid out in memory so gradient lookups hit cached blocks – avoids main-memory latency.
3. **Sparsity-aware.** Exploits the fact that many datasets are mostly zeros / missing; scans only the nonzero entries.
4. **Weighted quantile sketch.** For large datasets, pre-sorting every feature is expensive; XGBoost uses Hessian-weighted quantile buckets to propose splits quickly.

Same math (F1-F8), faster arithmetic.

Why these matter.

- On a 1M-row tabular dataset, naive boosting can take hours; XGBoost often finishes in minutes on a laptop
- *Sparsity-aware* is why XGBoost handles missing values natively (see next frame)
- *Weighted quantile* is Hessian-aware: bigger-curvature rows get finer split candidates

What does NOT change.

- F1 (additive model), F2 (objective), F3 (Taylor), F5 (leaf weight), F6 (Obj*), F7 (Gain), F8 (update) all identical to plain boosting
- Only the *search algorithm* inside a tree is engineered differently

Learn the math once; get the speed for free.

Parallel split / cache-aware / sparsity-aware / weighted quantile = engineering wins.

Missing Values: Default-Direction Split

What most models do. Fill missing values with a mean/median/mode BEFORE training. Called *imputation*. Adds noise and bias.

What XGBoost does. Learns a *default direction* (left or right) for each split. Rows with missing values for that split's feature ALL go the default way.

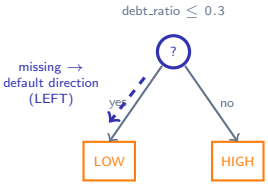
How it's learned. When evaluating a split, XGBoost tries both:

1. Send missing rows LEFT, compute Gain.
2. Send missing rows RIGHT, compute Gain.

Pick whichever direction gives higher Gain. Store as the split's *default direction*.

No imputation needed. The tree "learns where to route the gaps".

Picture.



Benefit. Missingness often carries signal ("this applicant didn't fill in the savings field"). XGBoost exploits it directly.

Natural handling of NAs is one of XGBoost's signature tricks.

Learn per-split default direction by trying both. No imputation step required.

Numeric Check: Squared Loss vs Log-Loss Head-to-Head

Same data point, two losses. Take $y = 1$ (positive), $\hat{y} = 0.3$.

Squared loss ($\ell = \frac{1}{2}(y - \hat{y})^2$).

$$g = -(y - \hat{y}) = -(1 - 0.3) = \mathbf{-0.7}.$$

$$h = 1.$$

Log-loss with $p = \sigma(\hat{y})$. Here $p = \sigma(0.3) \approx 0.574$.

$$g = p - y = 0.574 - 1 = \mathbf{-0.426}.$$

$$h = p(1 - p) = 0.574 \cdot 0.426 \approx \mathbf{0.245}.$$

Both g are negative (push $\hat{y}$ UP since $y = 1$), but log-loss pushes less hard.

Log-loss is curvature-aware: the 0.245 Hessian moderates the step.

Why the difference?

- Squared loss treats $\hat{y}$ as a raw number: Hessian $= 1$ everywhere. No probabilistic meaning.
- Log-loss treats $\hat{y}$ as a *logit*: Hessian varies with p, peaks at 0.25 when $p = 0.5$.

Implication for F5. Optimal leaf weight is $-G/(H + \lambda)$. Log-loss H is smaller (often < 1), so log-loss weights are *larger per unit gradient* – but tempered by the curvature when $p \to 0$ or 1.

Which to use?

- Binary classification $\Rightarrow$ log-loss (`binary:logistic`)
- Regression $\Rightarrow$ squared loss (`reg:squarederror`)
- Multi-class $\Rightarrow$ softmax log-loss

Same F3/F5/F7 machinery; different g_i, h_i recipes.

$y = 1$, $\hat{y} = 0.3$: **sq** $g = -0.7$, $h = 1$; **log** $g = -0.426$, $h = 0.245$. **Plug into F5 for leaf weight.**

Cross-Validation for XGBoost Tuning

Problem. We have three key hyperparameters to tune: K (trees), η (learning rate), `max_depth`. A single train/val split gives us one noisy estimate.

K-fold cross-validation. Split the training data into K equal folds. For each fold:

1. Train on the other $K-1$ folds.
2. Evaluate on the held-out fold.

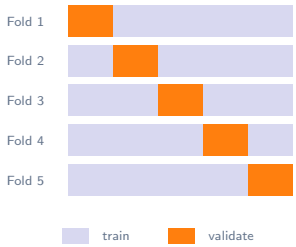
Average the K validation losses. This is the *CV loss*.

Recipe.

- For each candidate $(K, \eta, \texttt{max\_depth})$: compute CV loss.
- Pick the setting with the lowest CV loss.
- Retrain on the full training set.

CV = robust estimate of generalisation. More cost than single split, far less noisy.

5-fold picture.



Numeric. 5 folds, 3 depth values $\{3, 5, 7\}$: 15 models. Pick the depth with the lowest average val-loss.

xgboost.cv() does this natively with early stopping per fold.

K-fold CV = K train/val trials averaged. Pick lowest-loss setting; retrain on full data.