INSA ROUEN NORMANDIE

Machine Learning

Boosting: AdaBoost, Gradient Boosting and XGBoost

ITI 4 · Benoit Gaüzère

Plan for today

1. Why combine simple models?
2. **AdaBoost:** change which observations matter most.
3. **Gradient boosting:** learn the corrections still needed.
4. **XGBoost:** regularize and efficiently build boosted trees.
5. Choose tree complexity, learning rate and stopping.

Several imperfect models can help each other

A prediction can combine several models instead of relying on one.

Observation	True class	Model A	Model B	Model C	Majority
1	+1	+1	+1	−1	+1
2	−1	−1	+1	−1	−1
3	+1	−1	+1	+1	+1

Each model makes one error; here their majority makes none.

The models need useful predictions and errors that do not always coincide.

Bagging

In bagging (used by Random Forests, see previous lesson) :

1. Draw bootstrap training samples, with replacement.
2. Fit one model on each sample.
3. Average their predictions or combine their votes/probabilities.

$\Rightarrow$ The models can be trained separately.

$\Rightarrow$ Random forests also randomize the candidate features at each node.

Boosting: build the models in sequence

Principle: Fit a first simple model, inspect what it gets wrong, then adapt the next fit.

	Bagging	Boosting
How fits differ	Bootstrap samples	Information from earlier fits
Training order	Separate fits	Sequential fits
Main idea here	Stabilize predictions by averaging	Improve the current ensemble step by step

Main difference: For boosting decision trees, the tree m depends on previous trees $1, \ldots, m-1$

A small learner: the decision stump

A **stump** has depth 1: one test, two leaves.

It captures one simple pattern, such as “is $x_1 \leq 3.5$?”

Question	Intuition
Why use it?	A small, fast learner; later trees can supply other patterns
Is it required?	No. Boosting can use deeper trees or other suitable learners
Where is it common?	AdaBoost often uses stumps; gradient boosting often uses shallow, deeper trees

The ensemble gains flexibility by combining simple predictions.

Why combine weak learners?

A **weak learner** only needs to beat chance on the current **weighted** data ($\varepsilon_m < 0.5$ for binary AdaBoost):

- Simple learners are fast and have limited capacity individually.
- Later learners can capture patterns that the current ensemble misses.
- The ensemble gains complexity by adding learners, rather than making one tree very deep.

Deeper trees are allowed; their complexity is selected on development data.

Weak does not mean deliberately making wrong predictions.

Freund & Schapire (1997)

AdaBoost: give more attention to current mistakes

Keep the same inputs and true labels at every round.

1. Start with equal importance for all observations.
2. Fit a **new** classifier on (X, y) with `sample_weight` $= q^{(m)}$.
3. Increase the relative weights of its mistakes.
4. Fit the next classifier with the updated weights.
5. Combine the classifiers with a weighted vote.

First AdaBoost round: one mistake

Use labels −1/+1 and eight observations with initial weights $q_i = 1/8$.

x_i	1	2	3	4	5	6	7	8
True y_i	−1	−1	−1	+1	+1	−1	+1	+1
Stump $h_1(x_i)$	−1	−1	−1	+1	+1	+1	+1	+1

The learned split is $x \leq 3.5$.

Observation **6** is the only mistake.

Weighted error: what does the indicator select?

The indicator is **1 for a mistake, 0 for a correct prediction**:

$$\varepsilon_m = \sum_i q_i \, \mathbf{1}_{h_m(x_i) \neq y_i}.$$

For the previous stump, only observation 6 is wrong:

$$\left(\mathbf{1}_{h_1(x_i) \neq y_i}\right)_{i=1}^{8} = (0, 0, 0, 0, 0, 1, 0, 0).$$

$$\varepsilon_1 = \tfrac{1}{8}(0 + 0 + 0 + 0 + 0 + 1 + 0 + 0) = \tfrac{1}{8}.$$

The indicator selects which observation weights contribute to the error.

How much should this classifier count?

The **weighted error** determines the classifier's vote:

$$a_m = \frac{1}{2} \log \frac{1 - \varepsilon_m}{\varepsilon_m}.$$

Weighted error	Meaning	Vote / handling
$\varepsilon_m = 0$	Perfect on training data	Special stopping case; do not evaluate the log
$0 < \varepsilon_m < 0.5$	Better than chance	Finite, positive a_m
$\varepsilon_m = 0.5$	No advantage over chance	$a_m = 0$: no contribution
$\varepsilon_m > 0.5$	Worse than chance	Do not use as a positive vote

Here: $\varepsilon_1 = 1/8 \Rightarrow a_1 = \frac{1}{2} \log 7 \simeq 0.973.$

We want error below 0.5. Zero error is a success, handled separately.

How do observation weights change?

First multiply, then normalize:

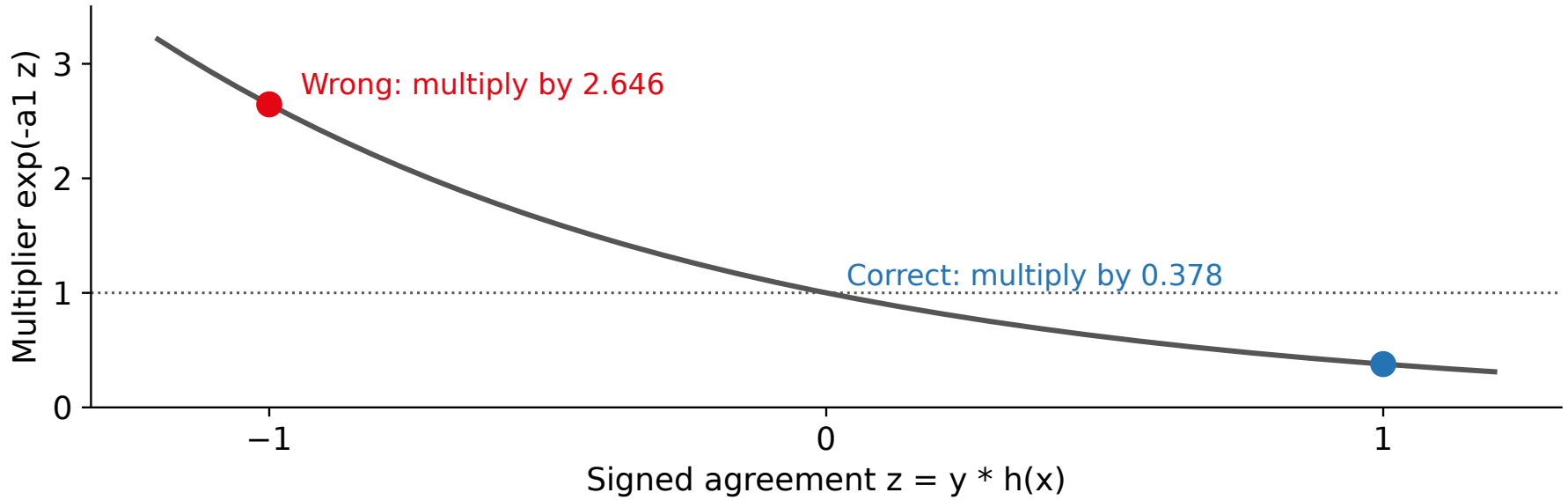
$$\tilde{q}_i = q_i \exp(-a_m y_i h_m(x_i)), \qquad q_i^{new} = \frac{\tilde{q}_i}{\sum_j \tilde{q}_j}.$$

Prediction	$y_i h_m(x_i)$	Multiplier before normalization
Correct	+1	$e^{-a_m} < 1$
Wrong	−1	$e^{a_m} > 1$

After round 1: point 6 has weight **1/2**; every other point has **1/14**.

Why the exponential emphasizes mistakes

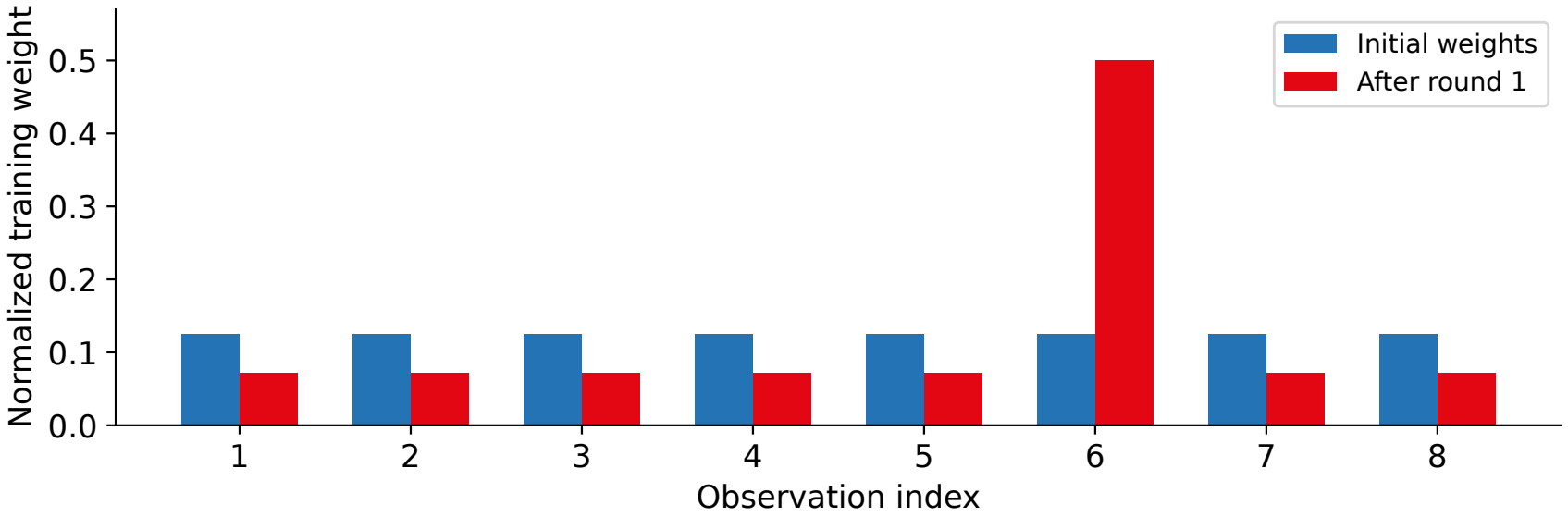
For $a_1 \simeq 0.973$, the multiplier is $e^{-a_1 z}$, where $z = y_i h_1(x_i)$.



A mistake has $z = -1$ and is amplified; a correct prediction has $z = +1$ and is reduced.

These factors act **before normalization**.

The next fit sees a different importance distribution



A **new stump is fitted with the updated observation weights**.

It now selects $x \leq 6.5$.

It corrects observation 6 but makes mistakes on 4 and 5.

Its weighted error is $2/14 = 1/7$, not the raw error fraction $2/8$.

Why does the second stump change its split?

The tree uses **weighted class proportions** to compare its candidate splits:

$$p_k(R) = \frac{\sum_{i:x_i \in R} q_i \mathbf{1}_{y_i = k}}{\sum_{i:x_i \in R} q_i}, \qquad G_q(R) = 1 - \sum_k p_k(R)^2.$$

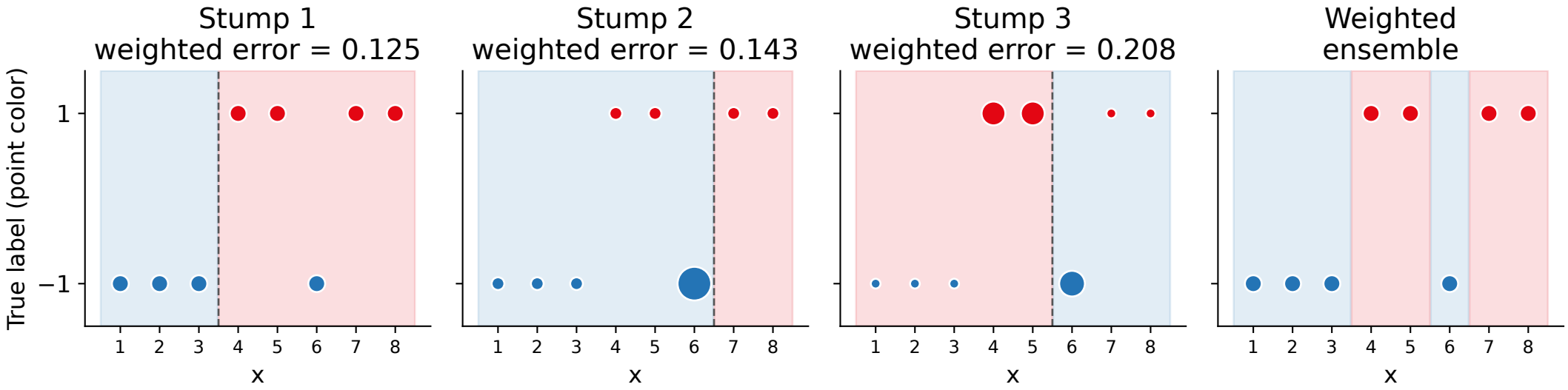
Candidate split	Child Gini, equal weights	Child Gini, updated weights
$x \leq 3.5$	**0.200**	0.364
$x \leq 6.5$	0.333	**0.238**

Child Gini is averaged using each child's **total observation weight**.

The preferred split changes. Leaf predictions use weighted majorities too.

q_i changes training; a_m controls the final vote.

Successive stumps and their ensemble



Point size shows the weight used by that round; background shows its prediction.

The final panel combines the three stumps rather than retaining only the last one.

Prediction is a weighted vote

- $h_m(x) \in \{-1, +1\}$: **which class** stump m votes for.
- $a_m > 0$: **how strongly** that stump's vote counts.

$$F(x) = \sum_{m=1}^{M} a_m h_m(x), \qquad \hat{y} = \operatorname{sign} F(x).$$

For point 6, the votes are $(+1, -1, -1)$:

$$F(6) \simeq 0.973 - 0.896 - 0.668 = -0.591.$$

The negative score selects class −1. All fitted stumps contribute.

Check-up 1 — Two different kinds of weight

After the first AdaBoost round:

1. Why is observation 6 more influential in the next fit?
2. The next stump misses observations 4 and 5, each of weight 1/14.
   What is its weighted error?
3. Which weights change the next split: q_i or a_m?
4. What would happen if all fitted stumps made exactly the same mistakes?

Check-up 1 — Answer

1. Its observation weight rose from 1/8 to 1/2 after a mistake.
2. The weighted error is $1/14 + 1/14 = 1/7$.
3. q_i changes weighted class proportions and the split criterion.
4. Repeating identical predictions would not correct the shared mistakes.

AdaBoost needs learners that respond usefully to the changing weights.

Noisy or wrongly labeled observations can receive increasing attention too.

Another way to improve the current model

AdaBoost changes **which observations matter most**.

Gradient boosting asks a different question:

what correction should the current prediction receive?

We will start with regression, where that correction is easy to calculate:

$$\text{residual} = \text{observed target} - \text{current prediction}.$$

Regression: the initial prediction

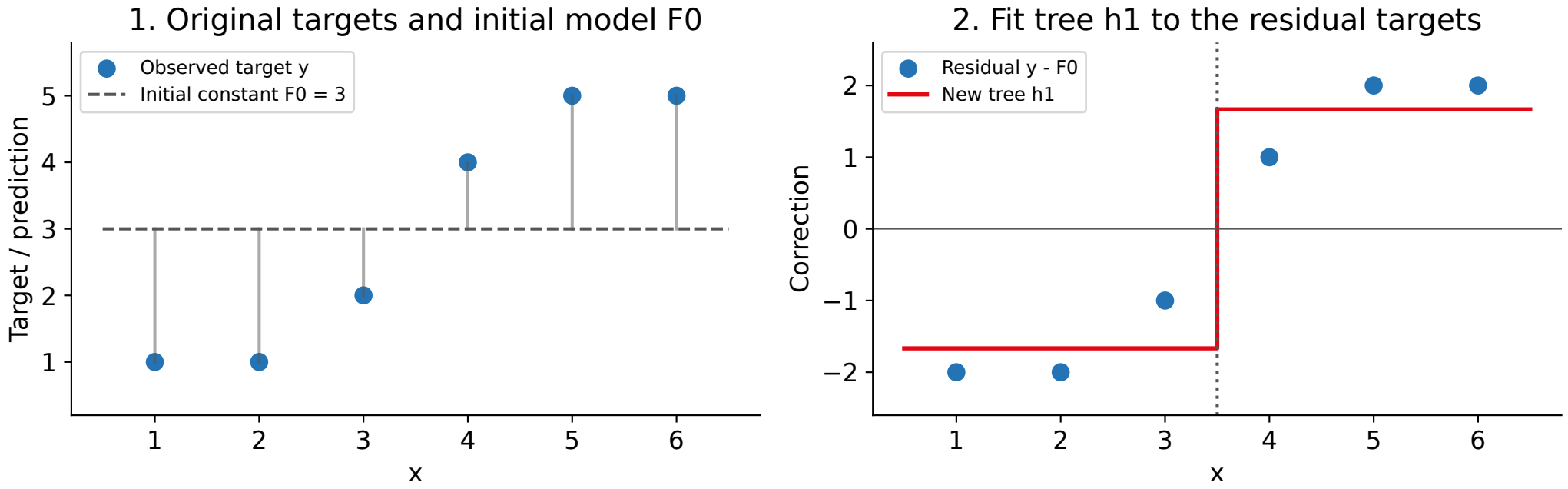
$F_m(x)$ is the **complete model prediction** after m correction trees.
F_0 is the initial model, before adding any correction tree.

For squared loss $\ell(y, F) = \frac{1}{2}(y - F)^2$, the best constant is $\overline{y} = 3$.

x_i	1	2	3	4	5	6
y_i	1	1	2	4	5	5
$F_0(x_i)$	3	3	3	3	3	3
$y_i - F_0(x_i)$	−2	−2	−1	+1	+2	+2

$F_0(x) = 3$ for every x: it can be represented by **one leaf with no split**.

Residuals become the next training target



Left: original targets y and the constant initial model $F_0 = 3$ (one leaf).

Right: residual targets $y - F_0$ and the first learned correction tree h_1 (two leaves).

The first correction tree

A regression stump selects $x \le 3.5$ and averages residuals in each leaf.

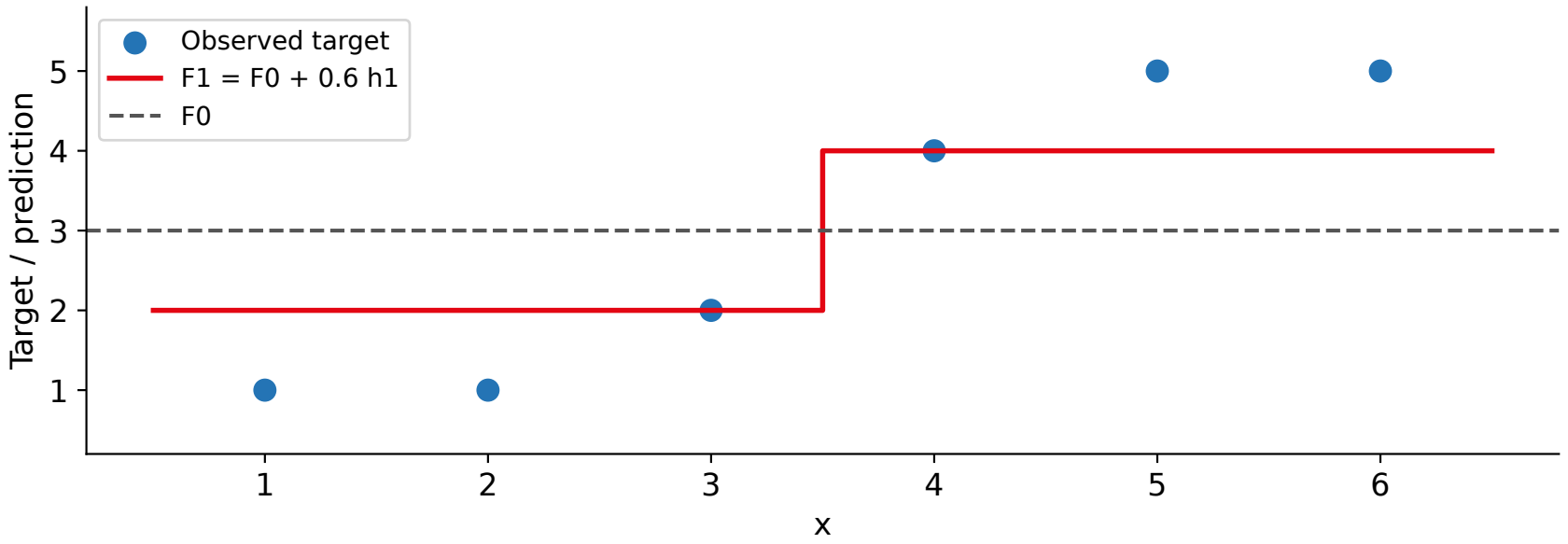
Leaf	Residuals	Tree output $h_1(x)$
$x \le 3.5$	$-2, -2, -1$	$-5/3$
$x > 3.5$	$+1, +2, +2$	$+5/3$

The **learning rate** ν is chosen by us, then tuned on development data.

Here $\nu = 0.6$: add **60%** of the learned correction.

$$F_1(x) = 3 + 0.6h_1(x) = \begin{cases} 2 & x \le 3.5, \\ 4 & x > 3.5. \end{cases}$$

The updated predictor



The new prediction is the **sum** of the initial value and a shrunken correction.

Residuals are recomputed after every update

x_i	1	2	3	4	5	6
y_i	1	1	2	4	5	5
$F_1(x_i)$	2	2	2	4	4	4
New residual $y_i - F_1(x_i)$	−1	−1	0	0	+1	+1

The second tree fits these **new** residuals.

$$\mathrm{MSE}(F_0) = 3, \qquad \mathrm{MSE}(F_1) = \frac{2}{3}.$$

Now we can name the additive model

$$F_m(x) = \underbrace{F_{m-1}(x)}_{\text{current full prediction}} + \nu \underbrace{h_m(x)}_{\text{new tree's correction}} .$$

For an input reaching leaf R_{jm}, the new tree returns its stored value v_{jm}:

$$h_m(x) = v_{jm} \quad \text{when } x \in R_{jm}.$$

- $h_m(x)$ is the correction **before** applying the learning rate.
- $\nu h_m(x)$ is the increment actually added.
- $F_m(x)$ is the resulting complete prediction or classification score.

Check-up 2 — The next correction target

At $x = 5$, the observed target is $y = 5$ and the current prediction is $F_1 = 4$.
A new leaf predicts a correction of $h_2 = 0.5$, with learning rate $\nu = 0.6$.

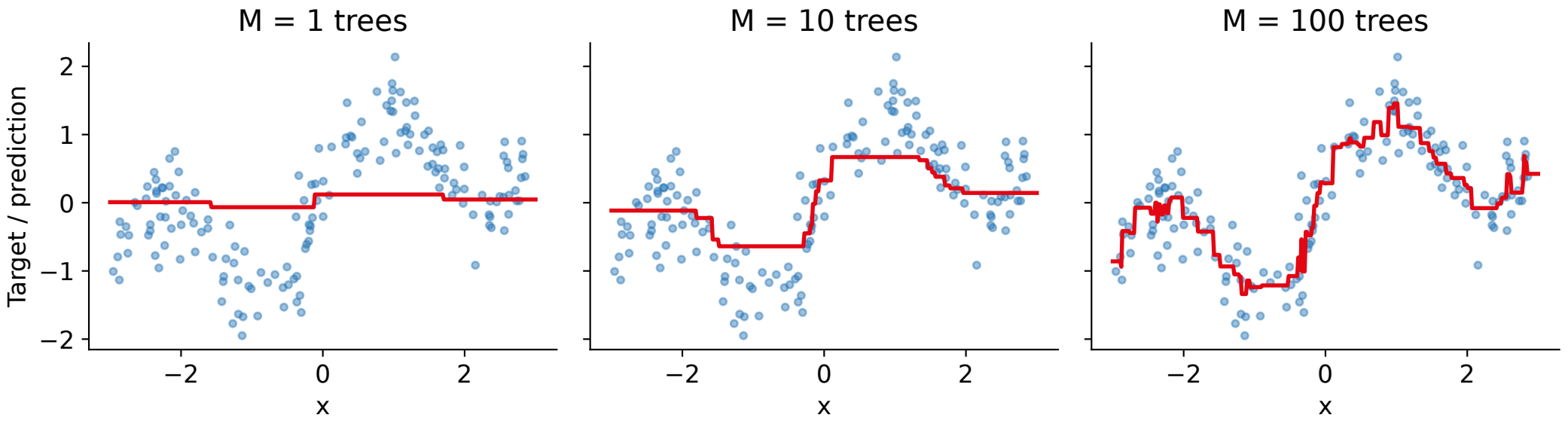
1. What training target did the new tree receive at this point?
2. What is the updated complete prediction?
3. What residual will the following tree receive there?

Check-up 2 — Answer

1. Tree target: $y - F_1 = 1$.
2. New prediction: $4 + 0.6 \times 0.5 = 4.3$.
3. New residual: $5 - 4.3 = 0.7$.

The residual at one point, the shared leaf correction and the full prediction are three different quantities.

Many small trees build a flexible function



Here the correction trees have **depth 2** (up to four leaves): they are **not stumps**.

Same data and learning rate 0.1; each curve is the full predictor after M stages.

Why is it called gradient boosting?

For squared loss:

$$\ell(y, F) = \tfrac{1}{2}(y - F)^2,$$

$$\frac{\partial \ell}{\partial F} = \tfrac{1}{2} \times 2(y - F) \times (-1) = F - y.$$

The **negative** derivative is $-(F - y) = y - F$: our ordinary residual.

General rule: fit trees to negative loss gradients.

Ordinary residuals $y - F$ are the special case of squared-loss regression.

Binary classification: from score to probability

Use labels $y \in \{0, 1\}$ and let $F(x) \in \mathbb{R}$ be a **score**.

The sigmoid converts it into a probability:

$$p = \sigma(F) = \frac{1}{1 + e^{-F}}, \qquad \sigma'(F) = \frac{e^{-F}}{(1 + e^{-F})^2} = p(1 - p).$$

For example, $F = 0$ gives $p = 0.5$; $F = \log 4$ gives $p = 0.8$.

To learn scores, use binary log loss:

$$\ell(y, p) = -y \log p - (1 - y) \log(1 - p).$$

Differentiate the loss with respect to the score

Chain the loss derivative and the sigmoid derivative:

$$\frac{\partial \ell}{\partial F} = \underbrace{\left(-\frac{y}{p} + \frac{1-y}{1-p}\right)}_{\partial \ell / \partial p} \underbrace{p(1-p)}_{\partial p / \partial F}.$$

$$= -y(1-p) + (1-y)p = p - y.$$

The next tree's target is the **negative** derivative:

$$r = -\frac{\partial \ell}{\partial F} = y - p.$$

For $y = 1$ and $p = 0.2$: the gradient is -0.8, the tree target is $+0.8$.

What does a score correction change?

Suppose the current score is $F_{m-1}(x) = 0$ and a learned leaf returns $h_m(x) = 1$.
Choose learning rate $\nu = 0.2$.

Quantity	Calculation	Value
Score increment	$\nu h_m(x)$	$+0.2$
New full score	$0 + 0.2$	0.2
Old probability	$\sigma(0)$	0.5
New probability	$\sigma(0.2)$	$\simeq 0.550$

We add **0.2 to the score**; the probability increases by about **0.05**.

XGBoost builds on the same correction principle

XGBoost is a gradient-boosted tree library (outside scikit-learn), not a new voting rule.

It still grows trees sequentially and adds their corrections.
It also provides:

- an objective with explicit penalties on tree complexity and leaf values;
- gradient and curvature information to choose corrections;
- efficient split search and implementation strategies.

XGBoost: loss plus a cost for tree complexity

A simplified objective is:

$$\mathcal{J} = \sum_i \ell(y_i, F(x_i)) + \sum_m \Omega(h_m), \qquad \Omega(h_m) = \gamma J_m + \frac{\lambda}{2} \sum_{j=1}^{J_m} v_{jm}^2.$$

- J_m: number of leaves in tree m.
- v_{jm}: the **constant output stored in leaf j**, so $h_m(x) = v_{jm}$ there.
- $\gamma \geq 0$: cost for an extra leaf; $\lambda \geq 0$: penalty on leaf values.

Our earlier left leaf had $v = -5/3$; with $\nu = 0.6$ it added -1 to F.

A penalty shrinks a leaf correction

Keep the same leaf: residuals $(-2, -2, -1)$, so $n_R = 3$ and $\sum r_i = -5$.
Fix $F_{previous} = 3$ and learning rate $\nu = 0.6$; vary λ.

$$\min_v \ \tfrac{1}{2}\sum_{i \in R}(r_i - v)^2 + \tfrac{\lambda}{2}v^2 \quad \Longrightarrow \quad v^* = \frac{\sum r_i}{n_R + \lambda}.$$

Chosen λ	Learned v^*	Increment νv^*	New F
0	$-5/(3+0) = -5/3$	−1	2
2	$-5/(3+2) = -1$	−0.6	2.4

γ is irrelevant to this value calculation because the leaf structure is fixed.

What remains common across the implementations

Method	What changes between stages?	What a tree contributes
AdaBoost	Observation weights	Class vote $h_m(x) \in \{-1, +1\}$, weighted by a_m
Gradient boosting	Negative-gradient targets	Real correction $h_m(x)$, multiplied by ν
XGBoost trees	Corrections under a regularized objective	Real correction with explicit penalties

For gradient boosting and XGBoost, $F_m = F_{m-1} + \nu h_m$.

In binary classification, $p_m = \sigma(F_m)$ is computed **after** that addition.

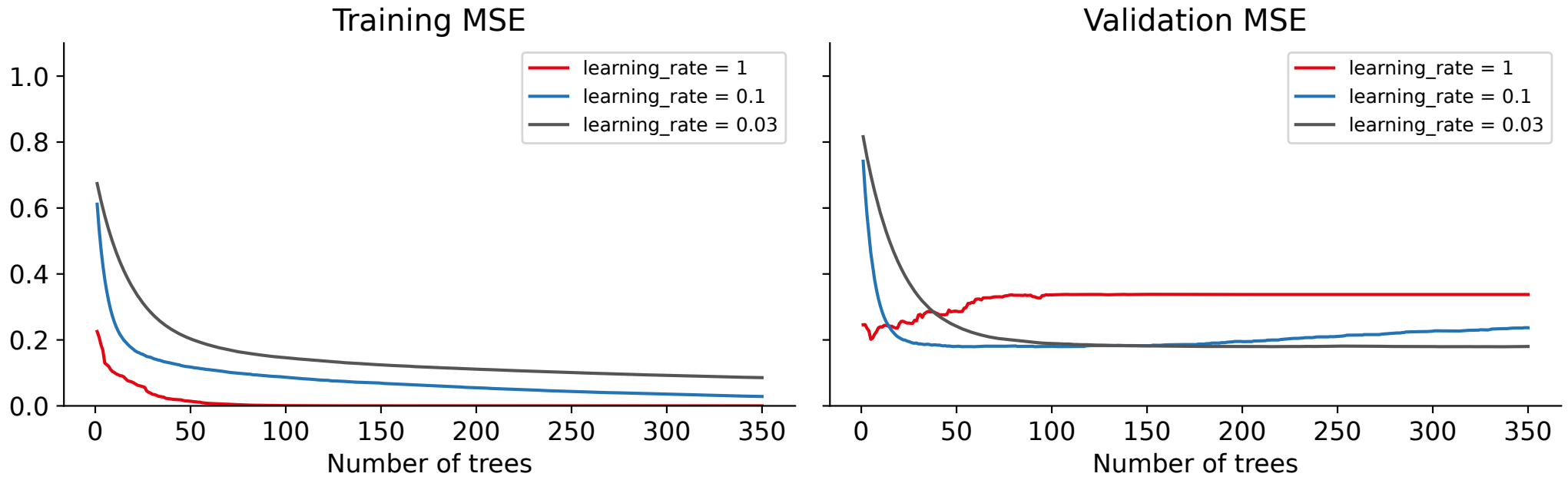
Parameters and the model equation

$$F_M(x) = F_0 + \underbrace{\nu}_{\texttt{learning_rate}} \sum_{m=1}^{\overbrace{M}^{\texttt{n_estimators}}} h_m(x).$$

Parameter	Constraint on the construction
`max_depth` $= D$	Each tree has at most 2^D leaves
`min_samples_leaf` $= L$	Each terminal region contains at least L fit observations
`subsample` $= s$	About $s\,n$ rows used per stage, without replacement

For XGBoost, the analogous row parameter is `subsample`; other names and rules can differ.

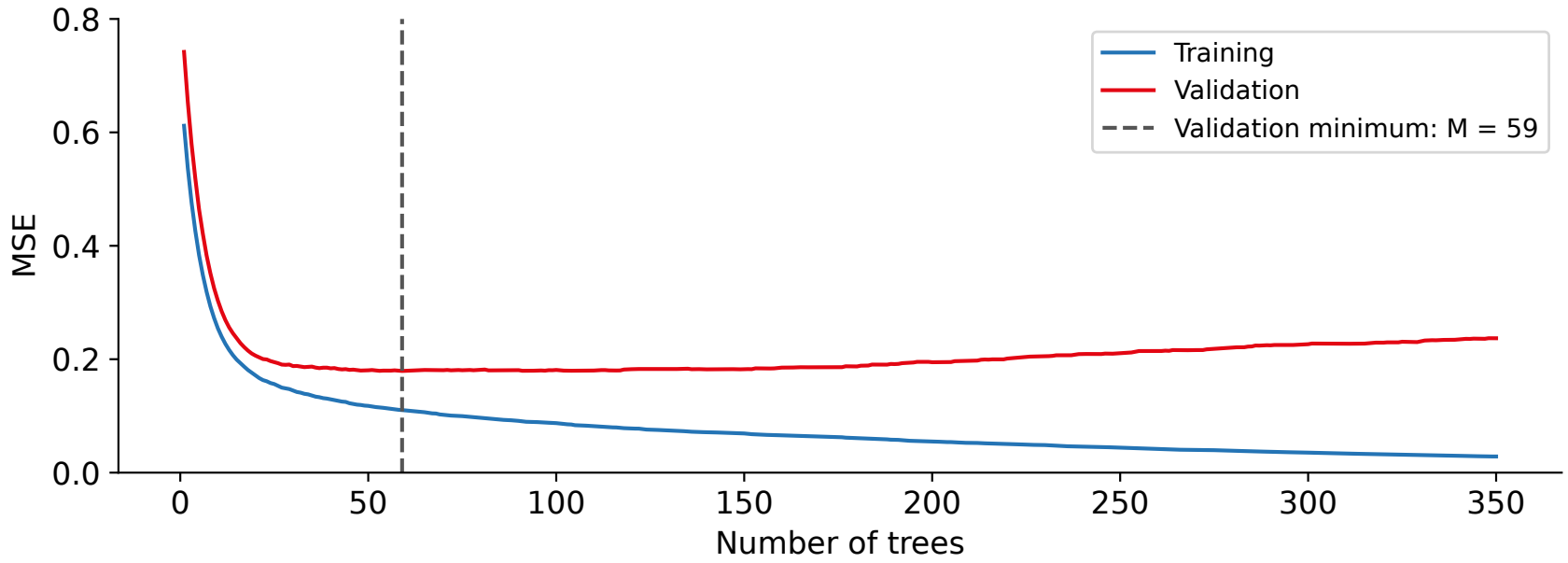
Learning rate and stage count work together



Same split and depth-2 trees; the curves use **MSE** throughout.

A slower training-loss decrease can accompany better validation performance.

Selecting the number of stages



Training loss keeps falling; validation loss reaches a minimum.

Use validation data to choose the stage count; keep test data for final evaluation.

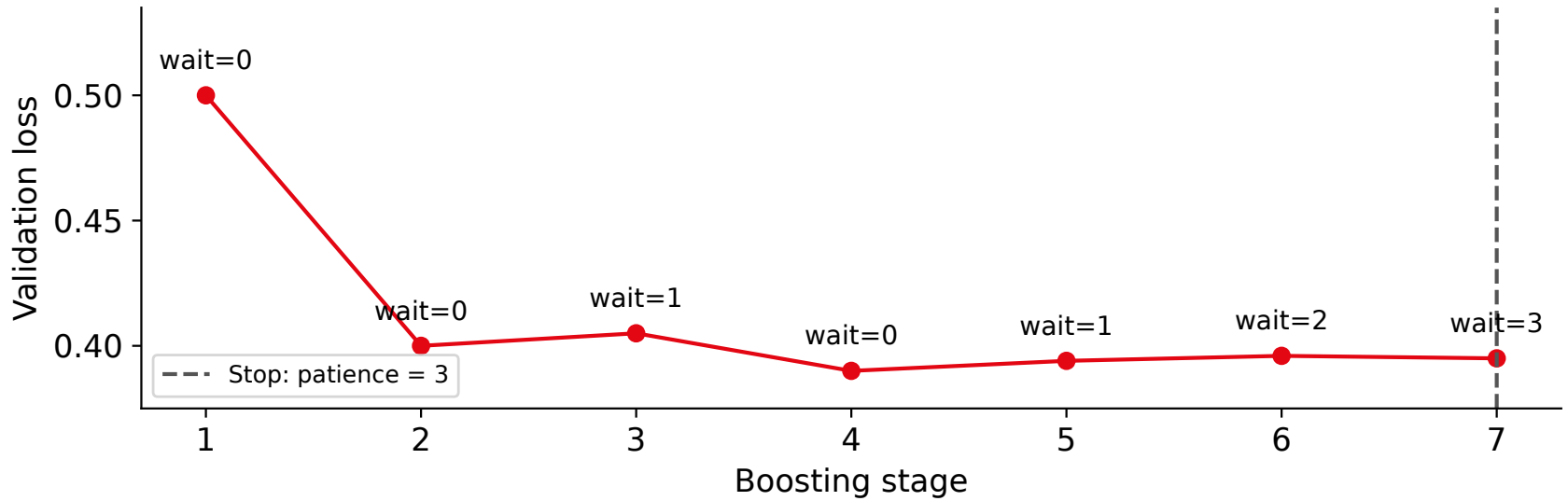
Early stopping: wait before giving up

Reserve validation observations and monitor their loss after each stage.

1. Keep a reference best validation loss.
2. If the loss improves by more than `tol`, reset the wait counter to zero.
3. Otherwise increase the counter.
4. Stop when the counter reaches the **patience** limit.

Keep test data out of this decision.

A patience counter through successive stages



Here patience is 3 and `tol=0.001`.

The improvement at stage 4 resets the counter; training stops at stage 7.

Best model is the stage 4

Check-up 3 — When should training stop?

Use patience 3 and `tol=0.001`.

Stage	1	2	3	4	5
Validation loss	0.50	0.40	0.405	0.402	0.406

What are the wait counts? When do we stop?

Which stage had the lowest loss?

Check-up 3 — Answer

The wait counts are **0, 0, 1, 2, 3**.

Stop after stage **5**.

Stage **2** had the lowest validation loss.

The loss at stage 4 fell relative to stage 3, but did not improve the best level.

Stopping iteration and best validation iteration are different concepts.

Early stopping in the sklearn implementation

```
model = GradientBoostingClassifier(
    n_estimators=500, learning_rate=0.05,
    max_depth=2, min_samples_leaf=5,
    validation_fraction=0.2,
    n_iter_no_change=10, tol=1e-4,
    random_state=42)
model.fit(X_dev, y_dev)
print(model.n_estimators_)
```

`n_iter_no_change` is the patience-like control; sklearn uses a rolling loss history.

A development and test protocol

1. Reserve the test set before selecting any model.
2. On development data, compare learning rate, tree size and stage budget by CV.
3. If early stopping is enabled, each CV training fold supplies its own stopping subset.
4. Freeze the selected procedure and refit on development data.
5. Evaluate the frozen models on test data once.

Select with the task metric; document the stopping loss separately.

Key ideas

- Boosting builds an ensemble **sequentially**.
- AdaBoost changes observation weights and combines class votes.
- Gradient boosting fits **negative gradients**; ordinary residuals are a squared-loss case.
- A new tree supplies a correction; the ensemble stores the complete prediction or score.
- Learning rate, tree size, penalties and stopping are selected on development data.

References and further reading

Raschka et al., *Machine Learning with PyTorch and Scikit-Learn*, chapter 7:
weak learners → AdaBoost → gradient boosting → XGBoost.

Freund & Schapire (1997): original AdaBoost paper

AdaBoost guide ·
Gradient boosting ·
XGBoost model

The following slides are optional mathematical or implementation details.

A second tree can choose another split

One optimal second stump selects $x \le 2.5$:

$$h_2(x) = \begin{cases} -1 & x \le 2.5, \\ +0.5 & x > 2.5. \end{cases}$$

x_i	1	2	3	4	5	6
$F_2 = F_1 + 0.6h_2$	1.4	1.4	2.3	4.3	4.3	4.3

MSE falls to approximately **0.247**.

A correction can worsen one observation while improving the total loss.

A small search on development data

```
search = GridSearchCV(
    GradientBoostingClassifier(
        random_state=42, n_iter_no_change=10),
    {"learning_rate": [0.03, 0.1],
     "max_depth": [1, 2],
     "n_estimators": [300]},
    scoring="roc_auc", cv=cv, n_jobs=1,
)
search.fit(X_dev, y_dev)
best_model = search.best_estimator_
```

Keep the same CV folds and score for the competing procedures.

Binary score derivatives and a Newton correction

For $p = \sigma(F)$ and binary log loss:

$$g_i = \frac{\partial \ell_i}{\partial F} = p_i - y_i, \qquad H_i = \frac{\partial^2 \ell_i}{\partial F^2} = p_i(1 - p_i).$$

A Newton correction for a fixed leaf is:

$$v_R \approx -\frac{\sum_{i \in R} g_i}{\sum_{i \in R} H_i}.$$

This is a score update, not a probability update.

XGBoost: a second-order leaf calculation

Write $G_R = \sum_{i \in R} g_i$ and $H_R = \sum_{i \in R} H_i$.

The regularized quadratic approximation gives:

$$v_R^* = -\frac{G_R}{H_R + \lambda}.$$

A split's approximate objective gain is:

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