Scientific Machine Learning 13
Ensemble Learning (Bagging and Boosting)

Donghyun Ko

January 4, 2026

What you will learn. This post will walk you through the full story of ensemble learning from its motivation to its modern algorithms. We will begin with the bias–variance dilemma and learn how combining multiple weak models can dramatically improve stability and accuracy. Then, we will dive into two families: **Bagging**, which fights variance by parallel averaging, and **Boosting**, which fights bias by sequential correction. By the end, you will understand why Random Forests and XGBoost dominate real-world machine learning tasks, and how every equation behind them connects intuitively to their behavior.

Contents

1 The Motivation & Intuition of Ensemble Learning

Why ensembles? An **ensemble method** combines several *base models* (often called *weak learners* when each one is only modestly accurate) to produce a single *strong learner* whose predictions are more accurate and robust than those of any individual model. Many flexible learners, like decision trees, k-nearest neighbors, and large-margin nonparametric models such as kernel SVMs, tend to be *low bias but high variance*: small perturbations of the training data can noticeably affect the fitted model. However, averaging many such predictors makes their idiosyncratic fluctuations cancel out, reducing variance.

Bias–variance Lens. Assume $Y = f(x) + \varepsilon$ with $\mathbb{E}[\varepsilon] = 0$ and $\mathrm{Var}(\varepsilon) = \sigma^2$. The expected squared prediction error at a point x is decomposed as

$$\mathbb{E}\big[(Y - \hat{f}(x))^2\big] = \underbrace{\Big(\mathbb{E}[\hat{f}(x)] - f(x)\Big)^2}_{\text{bias}^2} + \underbrace{\mathrm{Var}[\hat{f}(x)]}_{\text{variance}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

Only the first two terms depend on our learning algorithm; σ^2 is the *irreducible* part due to noise in the world.

Averaging reduces variance. Let $\hat{f}_1, \ldots, \hat{f}_M$ be M base predictors with common variance $\sigma^2_{\hat{f}}$ at x and average pairwise correlation ρ. Consider the average of the ensemble $\bar{f}_M(x) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}_m(x)$. Then,

$$\begin{aligned}\mathrm{Var}[\bar{f}_M(x)] &= \mathrm{Var}\left[\frac{1}{M}\sum_{m=1}^{M} \hat{f}_m\right] = \frac{1}{M^2}\left(\sum_{m=1}^{M} \mathrm{Var}[\hat{f}_m] + 2\sum_{1\le i<j\le M} \mathrm{Cov}(\hat{f}_i, \hat{f}_j)\right) \\ &= \frac{1}{M^2}\left(M\sigma^2_{\hat{f}} + 2\binom{M}{2}\rho\,\sigma^2_{\hat{f}}\right) = \left(\rho + \frac{1-\rho}{M}\right)\sigma^2_{\hat{f}}\end{aligned}$$

As M increases, the factor $\frac{1-\rho}{M}$ decreases to 0; as diversity increases (smaller ρ), the whole prefactor decreases. Thus, ensembles help most when (i) M is not tiny and (ii) base learners are *decorrelated*. This is exactly why bagging and random forests inject randomness (bootstrap samples, random feature subsets) to lower ρ.

> **Key motivation.** Combine multiple *weak learners / base models* trained on the same task to obtain a *strong learner* that reduces variance (via averaging) and/or reduces bias (via sequential refinement), while acknowledging that the noise term σ^2 cannot be learned away.

Two complementary motivations in practice.

- **Low-bias, high-variance families** (decision trees, kNN, complex-kernel SVMs, other flexible nonparametrics) benefit primarily from *parallel* aggregation that lowers variance: e.g., **bagging** and **random forests**.
- **High-bias, low-variance families** (linear / logistic regression, Naïve Bayes, other simple parametrics) benefit primarily from *sequential* additive modeling that lowers bias by focusing on previous errors: e.g., **boosting** (AdaBoost, Gradient Boosting, XGBoost).

Homogeneous vs. Heterogeneous Ensembles

- **Homogeneous**: all base learners share the *same* algorithm (e.g., many CART trees).
- **Heterogeneous**: base learners come from *different* algorithms (e.g., tree + SVM + logistic) and are combined.

Two execution patterns: parallel vs. sequential

1. **Parallel** methods build the base learners *simultaneously*; predictions are combined by *averaging/voting*. The primary goal is *variance reduction*. (e.g. bagging / random forests)
2. **Sequential** methods build base learners *progressively*, using feedback from earlier errors; later learners focus more on hard cases. The primary goal is *bias reduction*. (e.g. boosting)

Three major categories you will meet

1. **Bagging** (*homogeneous*, *parallel*): Train many weak learners on resampled data; combine their predictions by averaging/voting. Great for high-variance learners.
2. **Boosting** (*homogeneous*, *sequential*): Train learners one-by-one sequentially, each emphasizing previous residuals/misclassifications; combine their predictions by a deterministic weighting strategy at the end.
3. **Stacking** (*often heterogeneous,* parallel): Train a *meta-model* that combines predictions from diverse base models.

At a glance.

- **Bagging** ⇒ decorrelate via data randomness (bootstraps/features) ⇒ variance reduction via averaging decorrelated learners.
- **Boosting** ⇒ focus on mistakes via sequential reweighting/residuals ⇒ bias reduction via stage-wise fitting on residuals/misclassifications.
- **Stacking** ⇒ leverage complementary strengths of *different* model families via a learned combiner.
- **In common** ⇒ Irreducible noise remains.

2 Bagging

2.1 What is Bagging?

When we say **bagging** (Bootstrap AGGregatING), we really mean two simple actions: *make many slightly different training sets* through resampling the original data multiple times and then *combine the models* trained the single same model on them. The way we create those different training sets is called **bootstrapping**. In statistics, Bootstrapping1 is a statistical procedure that resamples a single dataset with replacement to create many simulated samples. This process allows us to calculate standard errors, construct confidence intervals, and perform hypothesis testing for sample statistics. Imagine that the original data are numbered $1, 2, \ldots, 12$. One bootstrap might look like $[1, 10, 3, 8, 3, 4, 2, 11, 9, 3, 5, 12]$: Here, you could notice the duplicates (3 appears three times) and the absences (e.g., 6 never shows up). The size of the sample can vary depending on the situation. Each such list produces a slightly different tree or classifier.

Bootstrap samples of size 5.

Suppose that our training set has n examples. We pull out n indices *with replacement*. Because we put each index back before drawing again, some examples appear twice (or more), and some do not appear at all. This new list of n indices defines one *bootstrapped sample*. If we repeat this M times, we get M different samples, written as $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(M)}$. In each of them, we train a base model $\hat{f}_m$.

Combining the models. Once the M base models are trained, we combine them at prediction time:

$$\hat{f}_{\text{bag}}(x) = \begin{cases} \frac{1}{M} \sum_{m=1}^{M} \hat{f}_m(x), & \text{regression (take the mean)}, \\ \text{majority vote of } \{\hat{f}_m(x)\}, & \text{classification (hard voting)}. \end{cases}$$

If the base models provide class probabilities, we may also *average the probabilities* and choose the largest (*This is called 'soft voting'*).

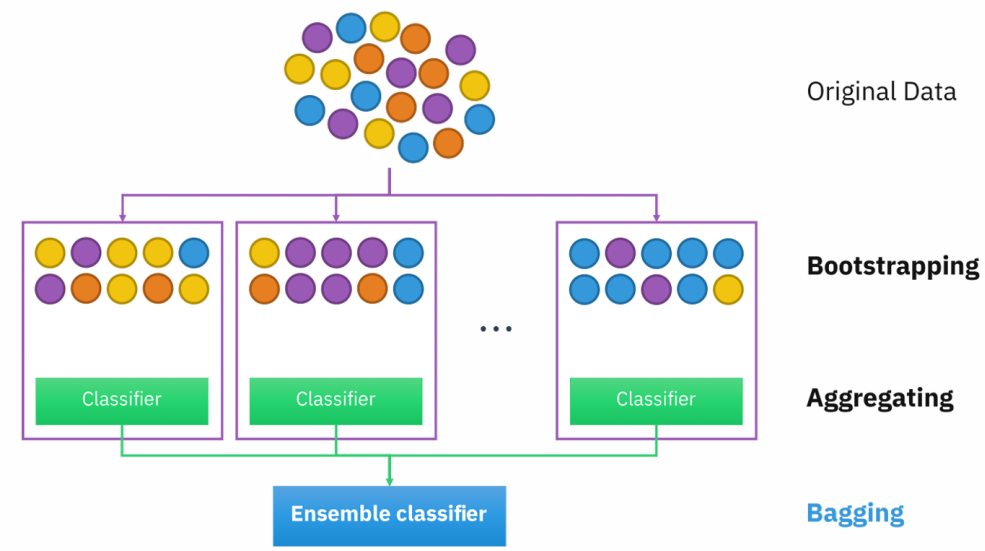

Illustration of Bagging.

Why does bagging help? Many powerful learners such as deep decision trees, kNN with small k, or flexible kernel methods are often *unstable*: a small change in the data can greatly change the fitted model (*high variance*). Because each bootstrap sample shakes the data in a slightly different way, the models make different small mistakes. Averaging these models cancels a large part of those fluctuations, so the final prediction becomes steadier. As a reminder, if each base predictor has variance $\sigma^2_{\hat{f}}$ and any pair has average correlation ρ, then

$$\mathrm{Var}\left[\frac{1}{M}\sum_{m=1}^{M}\hat{f}_m(x)\right] = \left(\rho + \frac{1-\rho}{M}\right)\sigma^2_{\hat{f}}$$

This expression tells a simple story: the *use of more models* ($M \uparrow$) and the *encouragement of diversity* ($\rho \downarrow$) reduce the variance of the ensemble.

Out-of-bag (OOB) estimation: a built-in test data. A pleasant surprise of bootstrapping is that each bootstrap sample *leaves out* some points. In fact, the chance that a particular point is not selected is $(1 - \frac{1}{n})^n \approx e^{-1} \approx 0.368$. Those left-out points are called the out-of-bag (OOB) set for that model. We can predict those OOB points using only the models that did not see them and then average those predictions across the models. The result is a nearly unbiased estimate of the test error, without running a separate cross-validation.

Summary. Bagging is best understood as a *variance* remedy. Averaging many decorrelated predictors drives down the variability of the final estimate (recall the factor $\rho + \frac{1-\rho}{M}$), so predictions become steadier across different samples. What bagging *does not* do is systematically bend the mean prediction toward the truth when all base models underfit in the same way—their average still carries that underfit, i.e., the *bias* is largely unchanged. This is why bagging naturally pairs with *unstable, low-bias/high-variance* learners (e.g. deep trees): it keeps their expressive power while taming their volatility. In short, **bagging is a variance cure, not a bias cure**.

2.2 Random Forest

Random Forest (RF) builds on the idea of *bagging*—training many decision trees on different bootstrap samples—and adds one more clever ingredient which is the **randomness in feature selection**. This small twist dramatically improves the performance and stability of decision tree ensembles.

Intuition. When we train many decision trees using bagging, each tree receives a different subset of the data due to the bootstrapping. However, if every tree always uses the same dominant feature to make its first few splits, the trees end up looking very similar behavior of the system. As a result, their predictions become somewhat highly correlated: when one tree makes an error, the others tend to make the same mistake. Finally, averaging those correlated trees does not reduce the variance much. Random Forest solves this problem by introducing an *additional layer of randomness*. At each node of every tree, instead of considering all p available features, the algorithm randomly picks only a subset of size m_{try} (commonly $m_{\text{try}} = \sqrt{p}$ for classification, or $m_{\text{try}} = p/3$ for regression). The best split is then chosen from just those features. This design ensures that not all trees "look at" the same features when making decisions. Some trees may focus on one subset of predictors, others on another. By doing so, **Random Forest** forces trees to explore different regions of the feature space, making their errors less correlated.

Why does this help? Reducing the correlation ρ between trees is the key to making ensembles powerful. Recall that the variance of an average of M base models is given by:

$$\text{Var}\left[\frac{1}{M}\sum_{m=1}^{M}\hat{f}_m(x)\right] = \left(\rho + \frac{1-\rho}{M}\right)\sigma_{\hat{f}}^2$$

This formula shows two levers for reducing ensemble variance:

- Increase M — use more trees.
- Decrease ρ — make the trees disagree more.

Bagging already increases M by resampling the data. Random Forest takes the second step by lowering ρ through random feature selection. And this is why we call it *'random forest'*, not just *'forest'*. The result is an ensemble that is both more diverse and stable. Another benefit is that because each split in the Random Forest only depends on a subset of features, the algorithm can easily handle datasets where some features are missing or noisy. A tree can simply skip those features at that node and rely on others, making the model more resilient to imperfect data.

Which trees to bag? In Random Forest, the individual decision trees are typically grown *deep* and left unpruned. Deep trees tend to have low bias but high variance—precisely the type of model that benefits most from averaging. When we average many such trees, their variances cancel out, while the bias remains low. By contrast, shallow trees (with high bias but low variance) are better suited for **boosting**, which focuses on reducing bias rather than variance.

Algorithm: Random Forest

1. For $m = 1, \ldots, M$ (number of trees):
   (a) Draw a bootstrap sample (with replacement) from the training data.
   (b) Grow a deep decision tree on this each sample:
      - At each node, randomly select m_{try} features from the p available.
      - Choose the best split only among these selected features.
2. Aggregate the M different results:
   - *Classification:* use majority voting (hard voting), or take the average of the probabilities (soft voting).
   - *Regression:* take the average of predictions.

To sum up, **Bagging** reduces variance by averaging multiple unstable learners trained on different bootstrap samples. **Random Forest** goes one step further: it adds randomness to feature selection at each split, making trees less correlated and the ensemble far more effective. It is, in essence, *bagging for trees with built-in feature diversity.*

3 Boosting

3.1 What is Boosting?

While **bagging** focuses on reducing variance by averaging many independent models, **boosting** takes the opposite approach — it aims to reduce *bias* by learning *sequentially.* The core idea is simple and elegant: instead of training many models in parallel, boosting trains them one after another, where each new model learns to fix the mistakes made by the previous ones. At the beginning, we start with a very simple model so-called *weak learner.* After evaluating its performance, we identify which training samples it predicted poorly. Then, we train a new model that gives *more weight* or *attention* to those difficult cases. By repeating this process many times, the ensemble gradually becomes stronger, since each new learner adds a small correction to the weaknesses of its predecessors.

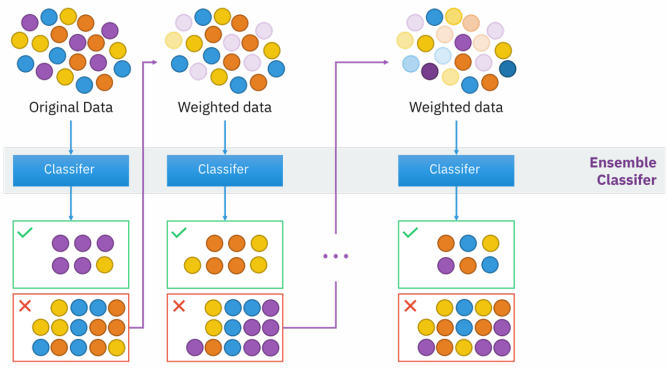

Illustration of Boosting.

How does Boosting work conceptually? Boosting is an **adaptive process**. The word "adaptive" means that the training procedure constantly adjusts itself depending on past performance. If certain data points were misclassified, boosting increases their importance (or weight) in the next round. This way, the algorithm "pays more attention" to what it previously got wrong. After several rounds, the final model is a weighted combination of many weak learners that together form a *strong predictor*. Intuitively, you can imagine a teacher (the algorithm) tutoring a student (the model). Each lesson focuses on the mistakes made in the last quiz — by repeatedly correcting errors, the student gradually masters the topic. That is precisely what boosting does with data.

How does Boosting work mathematically? Mathematically, we can express the final boosted model as an additive model:

$$F_M(x) = \sum_{m=1}^{M} \alpha_m h_m(x),$$

where each $h_m(x)$ is a weak learner trained at stage m, and α_m is the corresponding weight that determines how strongly this learner contributes to the overall prediction. To see why this is called *additive modeling*, consider that at iteration m, we already have a partially built model such that:

$$F_{m-1}(x) = \sum_{k=1}^{m-1} \alpha_k h_k(x),$$

which makes predictions that are not yet perfect. We then fit a new weak learner $h_m(x)$ to improve on what has already been built. Conceptually, the new learner is trained to approximate the *residual errors* of the previous stage:

$$r_i^{(m)} = y_i - F_{m-1}(x_i), \qquad i = 1, \ldots, n.$$

This means that $h_m(x)$ tries to predict where the current model $F_{m-1}(x)$ is wrong. Once $h_m(x)$ is trained, we update the ensemble as

$$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x),$$

where α_m (called the *learning rate* or *shrinkage* parameter) controls how much we trust the new correction h_m. If α_m is small, the model learns slowly but tends to generalize better.

Stagewise optimization view. Given data $\{(x_i, y_i)\}_{i=1}^n$ and a pointwise loss $\mathcal{L}(y, f)$, the goal of boosting is to find a function $F(x)$ that minimizes the empirical risk

$$\mathcal{R}(F) = \sum_{i=1}^{n} \mathcal{L}\Big(y_i, F(x_i)\Big).$$

Instead of searching for F directly, boosting builds it incrementally in a *forward stagewise* manner by adding one weak learner at a time from a hypothesis class $\mathcal{H}$ such as shallow decision trees.

(a) Initialization. The process begins with a simple initial model $F_0(x)$, which typically predicts the same value for all inputs. Formally, F_0 is chosen to minimize the overall loss:

$$F_0 \in \arg\min_{c \in \mathbb{R}} \sum_{i=1}^{n} \mathcal{L}(y_i, c).$$

For example, in squared-error regression ($\mathcal{L}(y, f) = (y - f)^2$), F_0 is simply the sample mean $\bar{y}$. This initial model captures the "baseline level" of the target variable, leaving subsequent stages to correct whatever patterns it fails to explain.

(b) Iterative updates. At each stage $m = 1, 2, \ldots, M$, the model is refined by adding a new weak learner $h_m(x) \in \mathcal{H}$ multiplied by a scaling factor α_m:

$$F_m(x) = F_{m-1}(x) + \alpha_m \, h_m(x).$$

The pair (α_m, h_m) is chosen to most reduce the current training loss:

$$(\alpha_m, h_m) = \arg \min_{\alpha \in \mathbb{R},\, h \in \mathcal{H}} \sum_{i=1}^{n} \mathcal{L}\Big(y_i,\; F_{m-1}(x_i) + \alpha \, h(x_i)\Big). \tag{S1}$$

Intuitively, $h_m(x)$ acts as a "correction term" that points toward the direction where the model currently performs poorly, while α_m determines how far we should move in that direction like a learning rate in a neural network.

(c) Squared-loss example (regression). In regression with squared loss, the residuals

$$r_i^{(m)} = y_i - F_{m-1}(x_i)$$

represent what the current model has not yet captured. Equation (S1) then reduces to a simple least-squares problem:

$$h_m \;\approx\; \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} \Big(r_i^{(m)} - h(x_i)\Big)^2.$$

Once h_m is fitted, the optimal step size α_m can be computed in closed form as

$$\alpha_m = \frac{\sum_{i=1}^{n} r_i^{(m)} \, h_m(x_i)}{\sum_{i=1}^{n} h_m(x_i)^2}.$$

Finally, the model is updated via $F_m = F_{m-1} + \alpha_m h_m$. In words, each stage fits the residuals left by the previous stage, applies a scaled correction, and repeats this process until the residuals are almost eliminated. This approach keeps learning stable—large steps risk overfitting, while smaller ones gradually refine the fit.

Why this closed form?

Under the squared loss, the stagewise objective with a fixed direction h_m is a simple one–dimensional least squares problem:

$$J(\alpha) = \sum_{i=1}^{n} \Big(r_i^{(m)} - \alpha\, h_m(x_i)\Big)^2, \quad \text{where } r_i^{(m)} = y_i - F_{m-1}(x_i).$$

Because $J(\alpha)$ is a convex quadratic in α, we can find its minimum analytically by differentiating:

$$\frac{dJ}{d\alpha} = -2\sum_{i=1}^{n} r_i^{(m)} h_m(x_i) + 2\alpha \sum_{i=1}^{n} h_m(x_i)^2 = 0,$$

which immediately gives

$$\boxed{\alpha_m = \frac{\sum_{i=1}^{n} r_i^{(m)} h_m(x_i)}{\sum_{i=1}^{n} h_m(x_i)^2}}$$

Geometric interpretation. This formula can be understood as an *orthogonal projection*. Minimizing $\|r^{(m)} - \alpha h_m\|_2^2$ means projecting the residual vector $r^{(m)}$ onto the one–dimensional subspace span$\{h_m\}$. Thus, α_m is exactly the projection coefficient:

$$\alpha_m = \frac{\langle r^{(m)}, h_m \rangle}{\langle h_m, h_m \rangle}, \quad \text{where } \langle f, g \rangle = \sum_i f(x_i) g(x_i).$$

If h_m is normalized so that $\|h_m\|_2 = 1$, this reduces to $\alpha_m = \langle r^{(m)}, h_m \rangle$.

(d) Classification example. When the task involves classification rather than regression, the same philosophy applies: each weak learner tries to correct the mistakes of the previous model. However, because the target labels are categorical (e.g., $y_i \in \{-1, +1\}$), we no longer use simple residuals but rather the *pseudo-residuals* which is the negative gradient of the chosen loss function with respect to F. Two of the most important cases are the exponential and logistic losses.

Example 1: Exponential loss (AdaBoost). In AdaBoost, the loss function is $\mathcal{L}(y, F) = \exp(-yF)$. This choice leads to a simple but powerful weighting mechanism. At iteration m, each training sample is assigned a weight

$$w_i^{(m)} = \exp\Big(-y_i F_{m-1}(x_i)\Big),$$

so that misclassified samples (where $y_i F_{m-1}(x_i) < 0$) receive higher importance. A new weak classifier $h_m(x) \in \{-1, +1\}$ is then trained to minimize the *weighted error rate*

$$\varepsilon_m = \frac{\sum_{i:\, y_i \neq h_m(x_i)} w_i^{(m)}}{\sum_i w_i^{(m)}}.$$

The optimal step size is given by

$$\alpha_m = \frac{1}{2} \log\Big(\frac{1 - \varepsilon_m}{\varepsilon_m}\Big),$$

and the model is updated as

$$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x).$$

After each iteration, the sample weights are adjusted:

$$w_i^{(m+1)} = w_i^{(m)} \exp\Big(- \alpha_m y_i h_m(x_i)\Big),$$

so that misclassified points gain even more influence in the next round. In this way, AdaBoost focuses attention on "hard" examples—those the current model struggles with—while reducing emphasis on correctly classified points.

Example 2: Logistic loss (LogitBoost / Gradient Boosting). Another popular loss for classification is the logistic loss

$$\mathcal{L}(y, F) = \log\Big(1 + \exp(-yF)\Big),$$

which corresponds to the negative log-likelihood of logistic regression. Here, the pseudo-residuals take the form

$$g_i^{(m)} = \frac{y_i}{1 + \exp\Big(y_i F_{m-1}(x_i)\Big)}.$$

Equivalently, if the labels are expressed as $y_i \in \{0, 1\}$, we can write

$$g_i^{(m)} = y_i - p_{m-1}(x_i), \qquad p_{m-1}(x_i) = \sigma\Big(F_{m-1}(x_i)\Big),$$

where σ is the logistic sigmoid function. At each stage, the weak learner $h_m(x)$ is trained to fit these pseudo-residuals, again using least squares:

$$h_m \approx \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} \Big(g_i^{(m)} - h(x_i)\Big)^2.$$

The optimal step size α_m is then found through a one-dimensional line search that minimizes the total loss. Finally, the model is updated as before:

$$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x).$$

Conceptually, h_m now learns to correct the model's miscalibrated probabilities rather than simple misclassifications. As iterations proceed, the predictions $F_M(x)$ converge to a logit scale, from which class probabilities can be recovered as $p_M(x) = \sigma(F_M(x))$.

Summary. Boosting, regardless of whether it is used for regression or classification, follows the same unifying principle:

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \alpha_m h_m(x),$$

where each new learner h_m aims to fix the residual or pseudo-residual—errors left by the previous ensemble. Regression focuses on reducing numerical discrepancies (residuals), whereas classification uses gradients of the loss to focus on points most likely to be misclassified. Through this sequential correction, a collection of weak learners is transformed into a single strong predictive model.

Advantages and disadvantages. Boosting is a very powerful approach with the points:

- It often **outperforms bagging** in both classification and regression problems.
- It is **provably effective** under certain theoretical assumptions.
- It is **versatile**, applicable to many loss functions and learning tasks.

However, it also has some drawbacks:

- It is **sensitive to outliers**, because each new model tries hard to correct every previous mistake.
- It is **hard to parallelize or scale**, since models are trained one after another.
- It can easily **overfit** if too many learners are added or if learning rates are too large.

Boosting vs. Bagging. It helps to view bagging and boosting as complementary ensemble methods:

- Bagging uses **random data resampling** to create uncorrelated trees trained in *parallel*, mainly reducing variance.
- Boosting uses **weighted data adjustment** to sequentially focus on hard examples, mainly reducing bias.

Hence, when the base model has **low bias but high variance**, bagging is the better choice. When the base model has **high bias but low variance**, boosting is typically more effective.

Popular variants of Boosting. Different boosting algorithms mainly differ in how they measure and correct errors:

- **AdaBoost (Adaptive Boosting):** adjusts sample weights to emphasize misclassified points.
- **Gradient Boosting:** fits each new learner to the residuals by minimizing a differentiable loss function.
- **XGBoost (eXtreme Gradient Boosting):** an efficient, regularized implementation of gradient boosting widely used in practice.
- Other variants include BrownBoost, LPBoost, and CoBoosting.

Takeaway. Boosting transforms a sequence of weak learners into a strong one by focusing each new model on the mistakes of its predecessors. Through this adaptive correction process, it gradually reduces bias and builds a highly accurate, though sometimes fragile, ensemble.

3.2 AdaBoost (Adaptive Boosting)

AdaBoost, short for *Adaptive Boosting*, is a pioneering boosting algorithm introduced by Freund and Schapire (1996). It is widely regarded as one of the most intuitive and effective ensemble methods — especially when decision trees (DTs) are used as weak learners.

Why "Adaptive"? The term "adaptive" reflects how AdaBoost dynamically adjusts the importance of training samples after each round. Samples that were misclassified by previous learners receive *higher weights*, forcing the next weak learner to pay more attention to these difficult cases. Meanwhile, correctly classified samples have their weights reduced, since the ensemble already handles them well. Through this mechanism, AdaBoost successively focuses learning effort on the mistakes of the ensemble — effectively "adapting" to its weaknesses.

Core procedure. The general idea of AdaBoost can be summarized as follows:

1. **Initialize sample weights.** Start with equal weights for all n samples:

$$w_i^{(1)} = \frac{1}{n}.$$

   These weights represent how much influence each observation has in fitting the next weak learner.

2. **Train a weak learner.** At iteration t, fit a base classifier $h_t(x)$ (e.g., a shallow decision tree) using the weighted data. Compute its weighted classification error:

$$\varepsilon_t = \sum_{i=1}^{n} w_i^{(t)} \mathbf{1}[h_t(x_i) \neq y_i].$$

   This error measures how poorly the weak learner performs under the current distribution of weights.

3. **Compute the learner's importance.** Assign a vote strength α_t to the weak learner based on its accuracy:

$$\alpha_t = \frac{1}{2} \ln \frac{1 - \varepsilon_t}{\varepsilon_t}.$$

   A learner with smaller ε_t (better accuracy) receives a larger α_t, meaning it contributes more strongly to the final decision.

4. **Update sample weights.** After evaluating h_t, update each sample's weight so that misclassified samples gain more influence in the next round:

$$w_i^{(t+1)} = w_i^{(t)} \exp(-\alpha_t y_i h_t(x_i)).$$

   Intuitively:

   - If $h_t(x_i) \neq y_i$, then $y_i h_t(x_i) = -1$, so $w_i^{(t+1)}$ increases (the model failed here).
   - If $h_t(x_i) = y_i$, then $y_i h_t(x_i) = 1$, so $w_i^{(t+1)}$ decreases (the model already got it right).

   The weights are then normalized to maintain $\sum_i w_i^{(t+1)} = 1$.

5. **Form the final classifier.** The final decision is made by taking a weighted vote of all the weak learners:

$$H(x) = \text{sign}\Big(\sum_{t=1}^{T} \alpha_t h_t(x)\Big).$$

Each h_t casts a "vote" proportional to its reliability α_t.

What makes AdaBoost powerful?

- **Adaptivity:** Each round focuses more on the mistakes of the current ensemble. The algorithm literally "adapts" its attention toward hard-to-classify samples.
- **Weight dynamics:** Misclassified samples are *penalized* (their weights increase), while correctly classified ones are *rewarded* (their weights decrease).
- **Cumulative voting:** Final decisions combine all base models, weighted by their performance. This produces a strong composite classifier from weak learners.

Theoretical insight. Although AdaBoost was not initially designed to minimize an explicit cost function, it can be shown to minimize the *exponential loss*:

$$L = \sum_{i=1}^{n} \exp(-y_i F(x_i)), \quad \text{where } F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Thus, AdaBoost can be interpreted as a stagewise additive model that iteratively finds the direction (weak learner) minimizing this exponential loss.

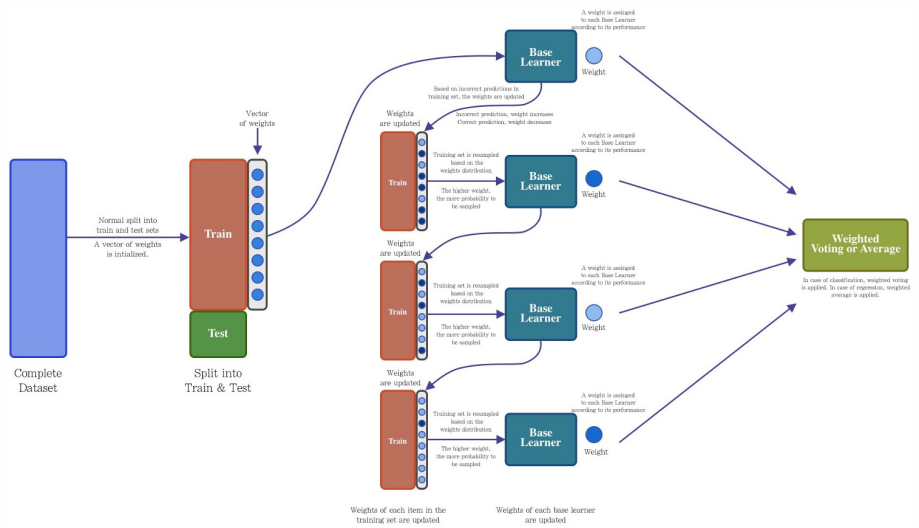

Illustration of the AdaBoost.

Intuitive analogy.

Think of each weak learner as a small flashlight trying to illuminate a dark room. The first light covers only a portion of the room, leaving some areas dark. AdaBoost then shines new lights precisely on those dark (misclassified) regions. As more lights are added, the room becomes brighter — until nearly all regions are correctly illuminated.

3.3 Gradient Boosting

While AdaBoost adjusts *sample weights* to emphasize misclassified points, **Gradient Boosting** instead adjusts the *model's predictions* directly by following the gradient of a chosen loss function. It can therefore handle *any differentiable loss* (squared error, logistic loss, etc.), not just the exponential loss used by AdaBoost.

Main idea. Gradient Boosting views boosting as an *optimization problem*: we want to find a function $F(x)$ minimizing the total loss

$$\mathcal{L}(F) = \sum_{i=1}^{n} \ell\Big(y_i, F(x_i)\Big).$$

Instead of updating weights like AdaBoost, Gradient Boosting adds weak learners in the direction that most decreases this loss. In other words, it performs **gradient descent in function space.**

Step-by-step intuition

1. **Compute gradients.** At iteration m, compute the **negative gradient** of the loss with respect to the model's current predictions:

$$r_i^{(m)} = - \left. \frac{\partial \ell(y_i, F(x_i))}{\partial F(x_i)} \right|_{F=F_{m-1}}$$

   This gradient acts like a *residual*: it points at the direction where the model is still poorly performing.

2. **Fit a weak learner.** Fit the new base model $h_m(x)$ to approximate these pseudo-residuals $\{r_i^{(m)}\}$:

$$h_m(x) \approx r_i^{(m)}.$$

3. **Find an optimal step size.** Determine how far to move along that direction by solving

$$\gamma_m = \arg\min_{\gamma} \sum_i \ell\Big(y_i, F_{m-1}(x_i) + \gamma\, h_m(x_i)\Big).$$

   This ensures that the update actually decreases the loss most efficiently.

4. **Update the model.** Add the scaled correction to the current model:

$$F_m(x) = F_{m-1}(x) + \nu\, \gamma_m h_m(x),$$

   where $\nu \in (0, 1]$ is the learning rate controlling how aggressively we move in that direction.

Gradient Boosting iteratively fits each new learner to the *residual errors* or, equivalently, the *gradients of the loss function.* In regression tasks, these residuals are simply $r_i = y_i - F_{m-1}(x_i)$. In classification tasks (with logistic loss), they become the difference between the true labels and predicted probabilities.

When trees are used as weak learners. If a Decision Tree (DT) is chosen as the base model, the resulting algorithm is called a **Gradient Boosted Tree (GBT)**, which in practice often *outperforms Random Forests*. The reason is that Gradient Boosting optimizes toward minimizing the residual error directly, whereas Random Forests simply average many uncorrelated trees.

Key difference from AdaBoost

- **AdaBoost:** identifies mistakes using sample weights — misclassified points get higher weights.
- **Gradient Boosting:** identifies mistakes using gradients of the loss function — residuals (errors) guide each step.

Regularization by design

- **Shrinkage (learning rate ν):** taking smaller steps helps prevent overfitting.
- **Subsampling:** using random subsets of data (Stochastic GBM) adds beneficial randomness, similar to Bagging.
- **Tree depth:** limiting tree size controls the complexity of each base learner.

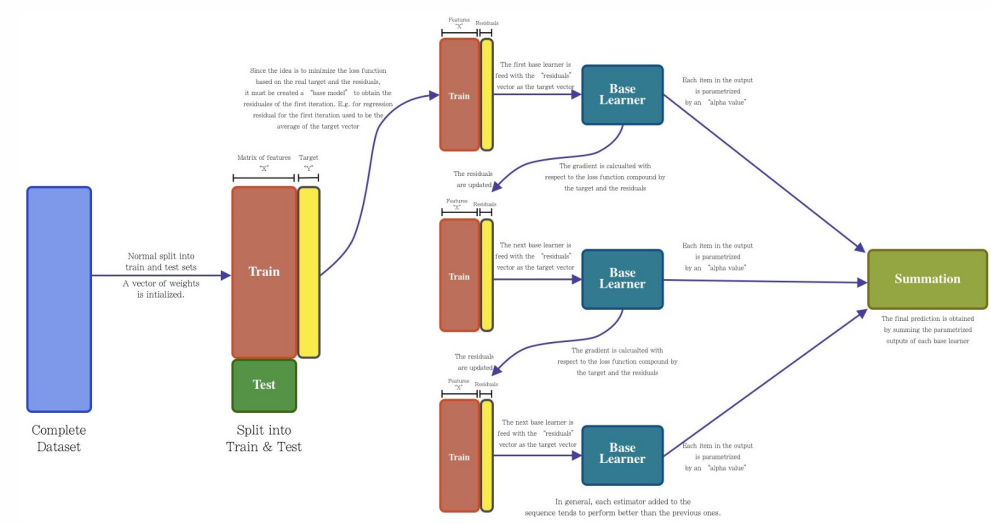

Illustration of Gradient Boosting.

3.4 XGBoost (Extreme Gradient Boosting)

While Gradient Boosting relies only on first-order information (gradients), **XGBoost** modernize the method by incorporating second-order optimization (using both gradient and curvature) and explicit regularization to control model complexity. These innovations made it one of the most powerful and widely used algorithms for structured/tabular data.

Objective function. At iteration t, the total objective combines the data-fitting loss with a penalty term that discourages overly complex trees:

$$\mathcal{L} = \sum_i \ell(y_i, \hat{y}_i) + \sum_k \Omega(f_k), \quad \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|_2^2,$$

where:

- f_k : the k-th decision tree (a weak learner),
- T : number of leaves (tree complexity),
- w : vector of leaf weights (predicted values),
- γ, λ : regularization parameters penalizing too many leaves or large weights.

Second-order Taylor expansion. To efficiently optimize $\mathcal{L}$, XGBoost performs a second-order Taylor approximation around the current predictions $\hat{y}_i^{(t-1)}$:

$$\ell(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) \approx \ell(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2,$$

where:

$$g_i = \frac{\partial \ell(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}, \qquad h_i = \frac{\partial^2 \ell(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)^2}}.$$

Thus, g_i measures how wrong the prediction is (slope), and h_i measures how confident or steep that wrongness is (curvature).

Optimization within each tree

For a fixed tree structure (partitioning of samples into leaf sets I_j), we can minimize the approximated objective over the leaf weights w_j. Grouping terms by leaf and setting the derivative to zero gives a closed-form optimum:

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}.$$

This elegant expression shows:

- The numerator aggregates the gradients — the "push" toward better predictions.
- The denominator adds curvature and regularization — the "resistance" to overfitting.

Evaluating tree splits. When deciding where to split a node, XGBoost computes the *gain* — how much the loss decreases if a split is made:

$$\text{Gain} = \frac{1}{2}\left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda}\right] - \gamma.$$

A larger gain indicates a more effective split. If the gain does not exceed γ, the split is pruned — this is how regularization directly shapes the tree.

Engineering innovations

Beyond its mathematical elegance, XGBoost is designed for *speed and scalability*. It introduces several system-level optimizations:

- **Histogram-based split finding**: speeds up tree building by discretizing features.
- **Sparse data optimization**: handles missing values and high-dimensional features.
- **Out-of-core computation**: enables training on datasets larger than memory.
- **Multi-threaded parallelism**: fully utilizes modern CPUs for training speed.

Why XGBoost dominates

XGBoost combines **theory** (regularized second-order boosting) with **engineering** (fast, scalable implementation). This balance of accuracy, interpretability, and efficiency explains why it remains the *workhorse algorithm* for structured data in competitions and real-world applications alike.

4 Conclusion

Bagging and Boosting are two opposite but complementary philosophies. Bagging focuses on *stability*: reduce variance by averaging many noisy models. Boosting focuses on *adaptation*: reduce bias by sequentially refining weak learners. Random Forests represent the maturity of Bagging — randomizing both samples and features to make trees independent and robust. Boosting, through AdaBoost, Gradient Boosting, and XGBoost, represents the art of turning weak learners into strong ones through iterative correction and regularization.

Key takeaways.

- Bagging → variance reduction through parallel randomness.
- Random Forest → Bagging + random feature subspace.
- Boosting → bias reduction through sequential correction.
- AdaBoost → exponential loss minimization by reweighting samples.
- Gradient Boosting → functional gradient descent for any loss.
- XGBoost → second-order optimization + regularization + scalability.

In the end, ensemble learning teaches an elegant lesson: a group of weak, diverse models — if properly organized — can collectively outperform even the smartest individual one.

This article is written for study purposes and independently elaborates on NE-795 (Scientific Machine Learning) lecture concepts by Prof. Xu Wu, NC State University.