Extreme Gradient Boosting (XGBoost)

Luu Minh Sao Khue

December 7, 2025

1. Key Ideas and Comparison to Gradient Boosting Trees

Gradient Boosting Trees (GBT) build a strong predictor by adding many weak regression trees in a stage-wise manner. At each boosting iteration, a new tree is fitted to the residual errors (or gradients) of the current model, and the final prediction is the sum of all tree outputs.

XGBoost (Extreme Gradient Boosting) follows the same general idea but improves it in three major ways:

- **Smarter math:** It uses both first- and second-order derivatives of the loss function with respect to the predictions, allowing Newton-style updates of tree leaf values and principled split scoring.
- **Stronger regularization:** It includes explicit regularization on tree complexity (number of leaves) and on leaf weights, reducing overfitting and encouraging simpler trees.
- **Efficient implementation:** It incorporates engineering tricks such as histogram-based split finding, column and row subsampling, and sparse-aware computation, which make it very fast and scalable.

Compared to a standard GBT implementation that typically uses only first-order information (gradients) and heuristic leaf values, XGBoost optimizes a regularized objective that directly includes second-order information and complexity penalties. This leads to more accurate trees, better generalization, and faster convergence.

2. Intuition Behind XGBoost

Consider a dataset with n samples. For each sample i, we have:

- feature vector $x_i \in \mathbb{R}^m$,
- target y_i,
- model prediction $\hat{y}_i$.

Instead of learning a single large model, gradient boosting builds the prediction function as a sum of many small regression trees:

$$\hat{y}_i^{(T)} = \sum_{t=1}^{T} f_t(x_i), \tag{1}$$

where f_t is the regression tree added at boosting iteration t. Each tree f_t is trained to *correct the mistakes* of the current model $\hat{y}^{(t-1)}$.

XGBoost improves this process by:

- using the first derivative (gradient) to know in which direction to correct the prediction,
- using the second derivative (Hessian) to know how confidently to move in that direction (how big the correction should be),
- choosing tree splits by how much they reduce a regularized approximation of the loss,
- computing the best leaf values in closed form, instead of by heuristic.

Thus each new tree is chosen to give the largest reduction in a regularized second-order Taylor approximation of the overall loss. The result is a sequence of small, carefully chosen corrections that gradually shape a strong and well-regularized predictor.

3. Mathematical Formulation

In this section we derive the XGBoost formulation step by step and explain all notation.

3.0.1 Notation and Regularized Objective

We have:

- n: number of training samples,
- m: number of features,
- T: number of boosting iterations (trees),
- $x_i \in \mathbb{R}^m$: feature vector of sample i,
- y_i: target/label of sample i,
- $\hat{y}_i^{(t)}$: prediction for sample i after t trees,
- f_t: regression tree added at iteration t.

The model after T trees is

$$\hat{y}_i^{(T)} = \sum_{t=1}^{T} f_t(x_i). \tag{2}$$

We define a regularized objective:

$$\mathcal{L} = \sum_{i=1}^{n} l\big(y_i, \hat{y}_i^{(T)}\big) + \sum_{t=1}^{T} \Omega(f_t), \tag{3}$$

where:

- $l(y_i, \hat{y}_i)$ is the loss for one sample (e.g., squared error or logistic loss),
- $\Omega(f_t)$ is the complexity penalty for tree f_t.

In XGBoost, a tree f is regularized as:

$$\Omega(f) = \gamma K + \frac{1}{2}\lambda \sum_{j=1}^{K} w_j^2, \tag{4}$$

where:

- K is the number of leaves (terminal nodes) in the tree,
- $w_j \in \mathbb{R}$ is the prediction value associated with leaf j,
- $\gamma \geq 0$ is a penalty per leaf (discourages too many leaves),
- $\lambda \geq 0$ is the L2 regularization on leaf values.

Improvement: Including γ and λ in the objective ensures that tree complexity is controlled during training, reducing overfitting and encouraging smaller, simpler trees.

3.1 Boosting Iteration and Second-Order Taylor Approximation

Boosting proceeds by adding trees one by one. Suppose we have already built trees $f_1, \dots, f_{t-1}$, giving predictions

$$\hat{y}_i^{(t-1)} = \sum_{k=1}^{t-1} f_k(x_i). \tag{5}$$

We now add a new tree f_t:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i). \tag{6}$$

The objective at iteration t can be written as

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l\big(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\big) + \Omega(f_t) + (\text{constant terms from previous trees}). \tag{7}$$

The terms from previous trees do not depend on f_t and can be ignored when optimizing f_t. We focus on the effective objective:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} l\big(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\big) + \Omega(f_t). \tag{8}$$

We approximate the loss using a second-order Taylor expansion with respect to the prediction. Let

$$g_i = \left.\frac{\partial l(y_i, \hat{y})}{\partial \hat{y}}\right|_{\hat{y}=\hat{y}_i^{(t-1)}}, \qquad h_i = \left.\frac{\partial^2 l(y_i, \hat{y})}{\partial \hat{y}^2}\right|_{\hat{y}=\hat{y}_i^{(t-1)}}, \tag{9}$$

where:

- g_i is the first derivative of the loss with respect to the prediction (gradient),
- h_i is the second derivative of the loss with respect to the prediction (Hessian).

For each sample i we approximate:

$$l\big(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\big) \approx l\big(y_i, \hat{y}_i^{(t-1)}\big) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2. \tag{10}$$

Summing over i and dropping the constant term $\sum_i l(y_i, \hat{y}_i^{(t-1)})$ we obtain the approximate objective:

$$\hat{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t). \tag{11}$$

Improvement: Using a second-order Taylor approximation allows XGBoost to include curvature information (via h_i) and to turn the problem of choosing leaf values into a simple quadratic optimization.

3.2 Tree Structure and Leaf Statistics

Each tree f_t is a piecewise constant function:

$$f_t(x) = w_{q(x)}, \tag{12}$$

where:

- $q(x)$ is the structure of the tree, mapping an input x to a leaf index $j \in \{1, \ldots, K\}$,
- w_j is the prediction value of leaf j,
- K is the number of leaves.

Define the index set of samples that fall into leaf j:

$$I_j = \{i \mid q(x_i) = j\}. \tag{13}$$

Then for $i \in I_j$ we have $f_t(x_i) = w_j$, and

$$\sum_{i=1}^{n} g_i f_t(x_i) = \sum_{i=1}^{n} g_i w_{q(x_i)} = \sum_{j=1}^{K} \left(\sum_{i \in I_j} g_i \right) w_j, \tag{14}$$

$$\sum_{i=1}^{n} \frac{1}{2} h_i f_t(x_i)^2 = \sum_{i=1}^{n} \frac{1}{2} h_i w_{q(x_i)}^2 = \sum_{j=1}^{K} \left(\sum_{i \in I_j} \frac{1}{2} h_i \right) w_j^2. \tag{15}$$

Define the aggregated gradient and Hessian for leaf j as:

$$G_j = \sum_{i \in I_j} g_i, \qquad H_j = \sum_{i \in I_j} h_i. \tag{16}$$

Plugging these into the objective:

$$\hat{\mathcal{L}}^{(t)} = \sum_{j=1}^{K} \left[G_j w_j + \frac{1}{2} H_j w_j^2 \right] + \gamma K + \frac{1}{2} \lambda \sum_{j=1}^{K} w_j^2. \tag{17}$$

We can group terms for each leaf:

$$\hat{\mathcal{L}}^{(t)} = \sum_{j=1}^{K} \left[G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma K. \tag{18}$$

Improvement: Summarizing each leaf by just (G_j, H_j) allows efficient computation of the objective and its minimizer, and enables fast split evaluation during tree construction.

3.3 Optimal Leaf Weights

For a fixed tree structure (i.e., fixed sets I_j), we optimize over leaf weights w_j. For each leaf j we consider:

$$\phi_j(w_j) = G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2. \tag{19}$$

Differentiating with respect to w_j and setting to zero:

$$\frac{d\phi_j}{dw_j} = G_j + (H_j + \lambda) w_j = 0, \tag{20}$$

gives the optimal leaf weight

$$w_j^* = -\frac{G_j}{H_j + \lambda}. \tag{21}$$

Interpretation:

- G_j is the total gradient in the leaf; $-G_j$ points in the direction that reduces the loss.
- $H_j + \lambda$ in the denominator scales the update by the curvature plus regularization. Larger curvature H_j or larger λ make the update smaller, stabilizing the model and shrinking leaf values.

Improvement: This closed-form solution provides an exact optimal update for leaf values under the quadratic approximation, in contrast to heuristic updates in plain GBT.

3.4 Score of a Tree Structure

Substituting w_j^* into $\hat{\mathcal{L}}^{(t)}$, we obtain the minimal value of the approximate objective for a given tree structure q:

$$\hat{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{K} \frac{G_j^2}{H_j + \lambda} + \gamma K. \tag{22}$$

This quantity (up to a constant) is used as the *score* of the tree structure. Lower values correspond to better (lower loss) trees.

3.5 Split Gain Formula

When growing a tree, XGBoost decides whether or not to split a node by computing the *gain* in the objective. Consider a node containing index set I, with:

$$G = \sum_{i \in I} g_i, \qquad H = \sum_{i \in I} h_i. \tag{23}$$

If this node is a leaf, its contribution to the objective is

$$\text{Score}_{\text{parent}} = -\frac{1}{2}\frac{G^2}{H + \lambda} + \gamma. \tag{24}$$

Suppose we split this node into left and right children with index sets I_L and I_R. Define:

$$G_L = \sum_{i \in I_L} g_i, \quad H_L = \sum_{i \in I_L} h_i, \qquad G_R = \sum_{i \in I_R} g_i, \quad H_R = \sum_{i \in I_R} h_i. \tag{25}$$

Then the contribution of the two children is

$$\text{Score}_{\text{children}} = -\frac{1}{2}\frac{G_L^2}{H_L + \lambda} - \frac{1}{2}\frac{G_R^2}{H_R + \lambda} + 2\gamma. \tag{26}$$

The gain from performing this split is defined as

$$\text{Gain} = \text{Score}_{\text{children}} - \text{Score}_{\text{parent}}. \tag{27}$$

After simplification this gives the standard XGBoost gain formula:

$$\boxed{\text{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda}\right) - \gamma}. \tag{28}$$

A split is accepted if Gain exceeds zero (or exceeds a user-specified minimum gain).

Improvement: The gain formula combines gradient and curvature information with regularization:

- The terms $\frac{G_L^2}{H_L+\lambda}$ and $\frac{G_R^2}{H_R+\lambda}$ reflect how much the loss can be reduced in each child, scaled by curvature and regularization.
- The term $\frac{G^2}{H+\lambda}$ represents the score of the unsplit node.
- The penalty γ ensures that small improvements do not lead to extra splits.

Thus XGBoost prefers splits that significantly reduce the regularized approximate loss and automatically controls tree complexity.

3.6 Examples of Gradients and Hessians

For completeness, we give two common examples of g_i and h_i.

Squared error regression. For loss

$$l(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2, \tag{29}$$

we have

$$g_i = \frac{\partial l}{\partial \hat{y}_i} = \hat{y}_i - y_i, \qquad h_i = \frac{\partial^2 l}{\partial \hat{y}_i^2} = 1. \tag{30}$$

Binary logistic loss. Let $\hat{y}_i$ be the logit and

$$p_i = \sigma(\hat{y}_i) = \frac{1}{1 + e^{-\hat{y}_i}}, \qquad y_i \in \{0, 1\}. \tag{31}$$

The loss is

$$l(y_i, \hat{y}_i) = -\left[y_i \log p_i + (1 - y_i)\log(1 - p_i)\right]. \tag{32}$$

Then

$$g_i = \frac{\partial l}{\partial \hat{y}_i} = p_i - y_i, \qquad h_i = \frac{\partial^2 l}{\partial \hat{y}_i^2} = p_i(1 - p_i). \tag{33}$$

These derivatives are plugged into the general framework above to compute G_j, H_j, optimal w_j^*, and the split gains.

4. Engineering Optimizations in XGBoost

In addition to its mathematical improvements, XGBoost derives much of its practical effectiveness from several system-level optimizations that greatly accelerate training. These engineering techniques make XGBoost scalable to large datasets and robust in real-world settings.

Histogram-based split finding. Instead of scanning every unique feature value when searching for the best split, XGBoost groups continuous features into histogram bins. Statistics are accumulated per bin, and splits are evaluated using these summaries. This reduces computation, improves memory locality, and allows multi-threaded parallel processing of features.

Column and row subsampling. XGBoost allows random subsampling of both features and data rows for each tree or even each split. This reduces overfitting, accelerates training, and increases model diversity—similar to the benefits seen in Random Forests.

Sparse-aware learning. Real-world datasets often contain missing or zero values. XGBoost treats missing values as informative and learns an optimal default direction (left or right) for them during split construction. This avoids manual imputation and makes the algorithm naturally efficient on sparse matrices.

Cache-aware and block-structured computation. Data are stored in a compressed column format with blocks arranged to maximize cache efficiency. This improves memory access patterns during split evaluation and yields substantial speedups, especially on large datasets with many features.

Out-of-core training. When the dataset does not fit into memory, XGBoost supports out-of-core training by using on-disk storage with prefetching and data compression. This enables the model to scale to very large datasets without requiring specialized hardware.

These engineering optimizations complement the second-order optimization and regularization framework, making XGBoost both highly accurate and extremely efficient in practice.

5. Summary

XGBoost extends classical Gradient Boosting Trees through a combination of mathematical innovations and engineering optimizations. On the algorithmic side, it:

- optimizes a regularized objective that penalizes both loss and tree complexity, leading to simpler and more robust models,
- uses second-order (gradient and Hessian) information to form a local quadratic approximation of the loss, enabling more stable and Newton-like updates,
- computes the optimal weight for each leaf in closed form, avoiding heuristic updates,
- selects splits using a gain formula that balances loss reduction against model complexity through explicit regularization.

In addition to these mathematical improvements, XGBoost incorporates several system-level optimizations that make it highly efficient in practice:

- histogram-based split finding and cache-aware data layouts that accelerate split evaluation,
- column and row subsampling that improve both speed and generalization,
- sparse-aware handling of missing values that eliminates the need for imputation,
- out-of-core training capabilities that scale to large datasets.

These advances make XGBoost more accurate, better regularized, faster, and more scalable than standard gradient boosting implementations.