# A Hobbyist's Notes on Gradient Boosting

**Gene Boo, 2023 — reviewed and reorganized edition**

> **Important note.** This is an informal, layman-friendly learning document by a hobbyist coder, not a formal mathematical treatise. It is intended to build intuition and practical understanding. Do not rely on it as a sole source for formal proofs, academic theses, production model validation, or regulated model approval.

## Table of Contents


- A Hobbyist's Notes on Gradient Boosting
  - Table of Contents
  - Preface: Scope, Learning Path, and Terminology
  - Part I — The Big Picture
    - 1. What Is Boosting?
    - 2. The Additive Boosting Model
    - 3. A First Regression Example
  - Part II — Boosting Compared with Logistic Regression
    - 4. Logistic Regression: The Transparent Baseline
    - 5. Boosting: The Nonlinear Challenger
    - 6. Logistic Regression vs Boosting: Practical Comparison
    - 7. Worked Credit-Risk Example
    - 8. Interpretability, Overfitting, Calibration, and Governance
  - Part III — Gradient Boosting Mathematics
    - 9. Gradient Boosting as Gradient Descent in Function Space
    - 10. Binary Classification with Logistic Loss
    - 11. Decision Trees as Weak Learners
    - 12. Regularization in Gradient Boosting
  - Part IV — XGBoost
    - 13. What Is XGBoost?
    - 14. XGBoost's Second-Order Taylor Approximation
    - 15. XGBoost Optimal Leaf Weight
    - 16. XGBoost Split Gain
  - Part V — LightGBM
    - 17. What Is LightGBM?
    - 18. Histogram-Based Split Finding
    - 19. Leaf-Wise Tree Growth

        - 20. Gradient-Based One-Side Sampling (GOSS)
        - 21. Exclusive Feature Bundling (EFB)
    - Part VI — CatBoost
        - 22. What Is CatBoost?
        - 23. Ordered Target Statistics
        - 24. Ordered Boosting
        - 25. Symmetric or Oblivious Trees
    - Part VII — Choosing and Using the Algorithms
        - 26. Conceptual Comparison
        - 27. Which Algorithm Should You Use?
        - 28. Hyperparameters Explained Simply
        - 29. Full Worked Mini-Example Across Algorithms
        - 30. Common Mistakes and Safeguards
        - 31. Final Practical Checklist
    - Part VIII — Historical Maturity Timeline
        - 32. XGBoost, LightGBM, and CatBoost Maturity Timeline
    - Part IX — Mathematical Summary and Glossary
        - 33. Mathematical Summary
        - 34. Glossary and Interpretation of Mathematical Formulae
    - Appendix A — Vectorization, BLAS, Distributed Computing, and GPUs
    - Bibliography and Source Notes
    - Closing Summary


## Preface: Scope, Learning Path, and Terminology

This document explains four closely related ideas:

- **Gradient Boosting** — the general learning principle: build a strong model by adding many weak learners that correct previous errors.
- **XGBoost** — an optimized, regularized, second-order gradient boosting system.
- **LightGBM** — a fast gradient boosting framework focused on histogram split finding, leaf-wise growth, Gradient-Based One-Side Sampling (GOSS), and Exclusive Feature Bundling (EFB).
- **CatBoost** — a gradient boosting framework designed especially for robust categorical-variable handling through ordered target statistics and ordered boosting.

### Recommended learning path

1. Start with the **big picture**: boosting as repeated correction.
2. Compare boosting with **logistic regression**, because this clarifies why boosting is powerful but less transparent.

3. Learn the **mathematical core**: gradients, pseudo-residuals, trees, regularization, and logistic loss.
4. Study the three major practical implementations: **XGBoost**, **LightGBM**, and **CatBoost**.
5. Finish with the **practical guide**, timeline, mathematical summary, glossary, and appendix on high-performance computing.

# Part I — The Big Picture

## 1. What Is Boosting?

### 1.1 Formal explanation

Boosting is an **ensemble learning** method. An ensemble combines many simple models into one stronger model. In gradient boosting, the final prediction function is an additive model:

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

where:

- $x$ is an input observation or feature vector.
- $F_M(x)$ is the final prediction after $M$ boosting rounds.
- $F_0(x)$ is the initial prediction.
- $f_m(x)$ is the weak learner added at boosting round $m$.
- $\eta$ is the learning rate, also called **shrinkage**.
- $M$ is the number of boosting iterations.

Most modern gradient boosting systems use **decision trees** as weak learners. Therefore, each $f_m$ is usually a small regression tree.

The learning goal is to minimize an empirical loss:

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

where:

- $n$ is the number of training observations.
- $y_i$ is the true target value for observation $i$.
- $F(x_i)$ is the model prediction for observation $i$.
- $L(y_i, F(x_i))$ is the loss for observation $i$.

### 1.2 Layman explanation

Imagine a student taking a practice exam. The first attempt is rough. The student then studies the mistakes, takes another attempt focused on those mistakes, and repeats. Each new attempt does not start from scratch; it corrects what the previous attempt got wrong.

Gradient boosting works in the same spirit:

1. Start with a simple prediction.
2. Measure the mistakes.
3. Build a small model that focuses on correcting those mistakes.
4. Add that correction to the previous model.
5. Repeat many times.

The final model is like a team of small experts, where each expert specializes in correcting the remaining errors of the team so far.

## 2. The Additive Boosting Model

The essence of boosting is:

$$\text{strong model} = \text{initial guess} + \text{many small corrections}.$$

A practical boosting update is:

$$F_m(x) = F_{m-1}(x) + \eta f_m(x), \qquad 0 < \eta \leq 1.$$

The learning rate $\eta$ controls how much of the new correction is accepted. A smaller $\eta$ makes learning more cautious. It usually requires more trees, but often improves generalization.

## 3. A First Regression Example

Suppose we want to predict house prices using one feature, floor area. The true prices are:

| House | True price |
|---:|---:|
| 1 | 100 |
| 2 | 150 |
| 3 | 200 |

For squared-error regression, a common initial prediction is the mean target:

$$F_0(x) = \bar{y} = \frac{100 + 150 + 200}{3} = 150.$$

| House | True value $y_i$ | Initial prediction $F_0(x_i)$ | Residual $y_i - F_0(x_i)$ |
|---:|---:|---:|---:|
| 1 | 100 | 150 | -50 |
| 2 | 150 | 150 | 0 |
| 3 | 200 | 150 | 50 |

A small tree is then trained to predict the residuals. Suppose the first tree predicts:

$$f_1(x_1) = -40, \qquad f_1(x_2) = 0, \qquad f_1(x_3) = 40.$$

If the learning rate is $\eta = 0.5$, the updated prediction is:

$$F_1(x_i) = F_0(x_i) + \eta f_1(x_i).$$

Therefore:

$$\begin{aligned} F_1(x_1) &= 150 + 0.5(-40) = 130, \\ F_1(x_2) &= 150 + 0.5(0) = 150, \\ F_1(x_3) &= 150 + 0.5(40) = 170. \end{aligned}$$

The model has moved closer to the true prices. It has not fully corrected the errors because the learning rate deliberately takes a smaller step to reduce overfitting.

# Part II — Boosting Compared with Logistic Regression

## 4. Logistic Regression: The Transparent Baseline

Logistic regression is a simple, interpretable linear classification model. For binary classification, it models the probability of class $1$ as:

$$p(y = 1 \mid x) = \frac{1}{1 + e^{-z}},$$

where:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

Equivalently, logistic regression assumes that the **log-odds** are linear in the features:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

### Layman explanation

Logistic regression draws a relatively simple decision boundary. It says:

> Each feature contributes a fixed amount to the final risk score.

For credit risk, one may think of it as:

$$\text{risk score} = \text{base risk} + \text{effect of income} + \text{effect of age} + \text{effect of leverage} + \text{effect of delinquency count}.$$

Each variable has a coefficient that is easy to explain.

### Logistic regression objective

Logistic regression minimizes binary cross-entropy loss:

$$\mathcal{L}(\beta) = -\sum_{i=1}^{n} \left[ y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right],$$

where:

$$p_i = \frac{1}{1 + e^{-(\beta_0 + \beta^\top x_i)}}.$$

The model directly learns coefficients $\beta_0, \beta_1, \ldots, \beta_p$. Each coefficient tells us how a feature changes the log-odds of the event.

## 5. Boosting: The Nonlinear Challenger

Boosting builds a model by adding many small models sequentially:

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \eta f_m(x).$$

For binary classification, the boosted raw score is often transformed into a probability through the sigmoid:

$$p(y = 1 \mid x) = \frac{1}{1 + e^{-F_M(x)}}.$$

Each new tree tries to correct the mistakes of the previous trees.

### Layman explanation

Boosting is more like a committee of small decision trees:

- The first tree makes a rough guess.
- The second tree fixes what the first tree got wrong.
- The third tree fixes what remains wrong.
- This continues many times.

So boosting says:

> Let many small models work together, each correcting the previous errors.

## 6. Logistic Regression vs Boosting: Practical Comparison

At a high level:

> **Logistic regression is a simple, transparent linear model. Boosting is a flexible ensemble method that combines many weak models, usually trees, to capture complex nonlinear patterns.**

| Dimension | Logistic Regression | Boosting / XGBoost / LightGBM / CatBoost |
|---|---|---|
| Basic form | Single linear model | Ensemble of many trees |
| Prediction | Probability via sigmoid | Score from many trees, often converted to probability |
| Main assumption | Linear relationship in log-odds | Few strict assumptions; learns nonlinear patterns |
| Feature interactions | Must be manually added | Automatically discovered by trees |
| Threshold effects | Poor unless engineered | Natural for tree splits |
| Feature engineering | Often required | Usually less required |
| Interpretability | High | Medium to low unless explained with SHAP, feature importance, PDP/ALE, or surrogate models |
| Overfitting risk | Lower, especially when regularized | Higher if not tuned carefully |
| Training speed | Usually very fast | Slower, especially with many trees, but optimized libraries are fast |
| Inference speed | Very fast | Fast, but usually slower than logistic regression |
| Small data | Often strong and stable | Can overfit if not controlled |
| Large data | Works well | Works well; LightGBM and XGBoost are especially scalable |
| Missing values | Usually need imputation | XGBoost, LightGBM, and CatBoost can often handle them more naturally |
| Categorical variables | Need encoding | CatBoost handles categoricals especially well |
| Probability calibration | Often good when correctly specified | May need calibration |
| Regulatory defensibility | Strong | Requires more explanation and governance |
| Best use | Transparent baseline or regulated model | High-performance predictive model |

## 7. Worked Credit-Risk Example

Suppose you want to predict whether a borrower defaults. Features include:

- income,
- debt-to-income ratio,
- age,
- number of missed payments,
- employment type,
- loan amount.

### 7.1 Logistic regression approach

A logistic model may learn:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1(\text{income}) + \beta_2(\text{debt ratio}) + \beta_3(\text{missed payments}).$$

It assumes each feature has a relatively smooth directional effect. For example:

- more missed payments increase default probability,
- more income decreases default probability,
- a higher debt ratio increases default probability.

This is excellent for explanation, audit, and regulatory discussion because each coefficient has a clean interpretation.

### 7.2 Boosting approach

A boosting model can learn rules such as:

```
If missed payments > 2
and debt ratio > 45%
and income < 4000
then increase default risk sharply.
```

It can also learn different behavior in different regions:

```
For young borrowers, income may matter strongly.
For older borrowers, debt ratio may matter more.
For high-income borrowers, missed payments may dominate.
```

Boosting does not need one simple straight-line relationship. It can discover complicated combinations automatically.

### 7.3 Numerical comparison

Suppose we have two features:

- $x_1 =$ income,
- $x_2 =$ debt ratio,

and target:

$$y = \begin{cases} 1, & \text{default}, \\ 0, & \text{no default}. \end{cases}$$

Assume the logistic regression score is:

$$z = -2 - 0.0001x_1 + 0.05x_2.$$

For a borrower with $x_1 = 3000$ and $x_2 = 50$:

$$
\begin{aligned}
z &= -2 - 0.0001(3000) + 0.05(50) \\
&= -2 - 0.3 + 2.5 \\
&= 0.2.
\end{aligned}
$$

So:

$$
p = \frac{1}{1 + e^{-0.2}} \approx 0.5498.
$$

Logistic regression predicts a default probability of about $54.98\%$.

A boosting model may instead use rule-based corrections:

```
Initial raw score: F_0 = 0
Tree 1: if debt ratio > 40, add 0.8
Tree 2: if income < 4000, add 0.5
Tree 3: if debt ratio > 60, add another 0.7
```

For this borrower, the first two rules apply but the third does not:

$$
F(x) = 0 + 0.8 + 0.5 + 0 = 1.3.
$$

Converted to probability:

$$
p = \frac{1}{1 + e^{-1.3}} \approx 0.7858.
$$

Boosting predicts about $78.58\%$ default probability because it captured threshold-style rules.

## 8. Interpretability, Overfitting, Calibration, and Governance

### 8.1 Linearity versus nonlinearity

Logistic regression is linear in the log-odds:

$$
\log\left(\frac{p}{1-p}\right) = \beta^\top x.
$$

A one-unit change in a feature has a consistent additive effect on the log-odds unless transformations or interaction terms are manually added. For example, if age is included as $x_1$, the model assumes:

$$
\text{effect of age} = \beta_1 x_1.
$$

If real-world risk is high for very young borrowers, lower for middle-aged borrowers, and higher again for elderly borrowers, logistic regression will not naturally capture that unless we add features such as $\text{age}^2$ or age buckets.

Boosting trees naturally capture nonlinearities and interactions:

```
If age < 25:
    use one risk rule
Else if age is between 25 and 60:
    use another risk rule
Else:
    use another risk rule
```

### 8.2 Coefficient interpretation in logistic regression

If the model is:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1,$$

then a one-unit increase in $x_1$ changes the log-odds by $\beta_1$. The odds multiplier is:

$$e^{\beta_1}.$$

If $\beta_1 = 0.2$, then:

$$e^{0.2} \approx 1.221.$$

This means a one-unit increase in $x_1$ multiplies the odds by about $1.221$, or increases the odds by about $22.1\%$.

### 8.3 Why boosting is harder to explain

Boosting may contain hundreds or thousands of trees. Its prediction is not one simple equation. Interpretation usually requires tools such as:

- feature importance,
- partial dependence plots,
- accumulated local effects,
- SHAP values,
- surrogate models.

Boosting can be explained, but not as directly as logistic regression.

### 8.4 Overfitting comparison

Regularized logistic regression can minimize:

$$\sum_{i=1}^{n} L(y_i, \hat{y}_i) + \lambda \sum_{j=1}^{p} \beta_j^2,$$

where the penalty term:

$$\lambda \sum_{j=1}^{p} \beta_j^2$$

discourages coefficients from becoming too large.

Boosting can overfit if:

- there are too many trees,
- trees are too deep,
- the learning rate is too high,
- leaves are too small,
- there is leakage,
- validation is not properly designed.

Boosting is controlled using:

- learning rate,
- maximum depth,
- number of leaves,
- minimum child weight,
- subsampling,
- column sampling,
- L1/L2 regularization,
- early stopping.

### 8.5 When each approach is better

Use **logistic regression** when:

- interpretability is more important than maximum accuracy,
- coefficient-level explanations are required,
- the problem is regulated or audit-heavy,
- the relationship is roughly linear,
- data is limited,
- a simple baseline model is needed,
- stable, monotonic, easy-to-defend behavior matters.

Example use cases include traditional scorecards, medical risk models requiring simple explanations, regulatory credit models, quick baseline classifiers, and odds-ratio analysis.

Use **boosting** when:

- predictive accuracy is the priority,
- relationships are nonlinear,
- feature interactions are important,
- features have mixed types,
- there is enough data,
- explainability tools can be used,
- strong tabular-data performance is needed.

Example use cases include fraud detection, credit default prediction challenger models, customer churn prediction, click-through-rate prediction, demand forecasting, ranking, pricing models, and operational-risk classification.

### 8.6 Baseline-and-challenger workflow

A common practical workflow is:

1. Build a **logistic regression baseline**.
2. Build a **boosting model challenger**.
3. Compare AUC, log loss, accuracy, precision/recall, calibration, stability, interpretability, and out-of-time validation performance.
4. Decide whether the performance gain from boosting justifies the added complexity.

This is especially useful in risk modelling because logistic regression is transparent, while boosting may deliver stronger predictive performance.

### 8.7 Calibration

Calibration concerns whether predicted probabilities correspond to observed event rates. For example, a calibrated predicted probability of $0.20$ should roughly correspond to a true event rate of $20\%$.

Logistic regression often gives reasonably calibrated probabilities if the model is correctly specified and validation is sound.

Boosting may rank observations very well but sometimes produce probabilities that need calibration before being interpreted literally. Common calibration methods include:

- Platt scaling,
- isotonic regression,
- beta calibration.

# Part III — Gradient Boosting Mathematics

## 9. Gradient Boosting as Gradient Descent in Function Space

### 9.1 Formal explanation

In ordinary gradient descent, we update parameters such as $\theta$:

$$\theta_m = \theta_{m-1} - \eta \nabla_\theta \mathcal{L}(\theta_{m-1}).$$

Gradient boosting follows a similar idea, but instead of directly updating a finite parameter vector, it updates an entire prediction function $F$.

At iteration $m$, compute the negative gradient of the loss with respect to the current prediction:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}.$$

The value $r_{im}$ is called a **pseudo-residual**. A new tree $f_m$ is trained to approximate these pseudo-residuals:

$$f_m \approx \arg\min_f \sum_{i=1}^{n} \big(r_{im} - f(x_i)\big)^2.$$

Then update:

$$F_m(x) = F_{m-1}(x) + \eta f_m(x).$$

### 9.2 Layman explanation

The word **gradient** means direction of steepest increase. Therefore, the **negative gradient** points in the direction that reduces the loss most quickly.

For squared error, the negative gradient is simply the ordinary residual. For more complex losses, such as logistic loss, the negative gradient is a mathematically generalized version of the residual.

So instead of saying:

> Fit the next tree to the errors.

Gradient boosting says more generally:

> Fit the next tree to the direction that most reduces the chosen loss function.

### 9.3 Squared-error gradient example

For squared-error loss:

$$L(y_i, F(x_i)) = \frac{1}{2}\big(y_i - F(x_i)\big)^2.$$

Differentiate with respect to $F(x_i)$:

$$\frac{\partial L}{\partial F(x_i)} = -\big(y_i - F(x_i)\big).$$

Therefore, the negative gradient is:

$$r_i = -\frac{\partial L}{\partial F(x_i)} = y_i - F(x_i).$$

This confirms that, for squared-error regression, pseudo-residuals are the usual residuals. If $y_i = 100$ and $F(x_i) = 150$, then:

$$r_i = 100 - 150 = -50.$$

The next tree should learn a negative correction for this observation.

## 10. Binary Classification with Logistic Loss

For binary classification, let $y_i \in \{0, 1\}$. The model often produces a raw score $F(x_i)$, also called a **logit**. This raw score is converted into a probability using the logistic sigmoid function:

$$p_i = \sigma(F(x_i)) = \frac{1}{1 + e^{-F(x_i)}}.$$

The binary logistic loss is:

$$L(y_i, F(x_i)) = -y_i \log(p_i) - (1 - y_i) \log(1 - p_i).$$

The first derivative with respect to the raw score is:

$$g_i = \frac{\partial L}{\partial F(x_i)} = p_i - y_i.$$

The negative gradient is:

$$r_i = y_i - p_i.$$

The second derivative is:

$$h_i = \frac{\partial^2 L}{\partial F(x_i)^2} = p_i(1 - p_i).$$

### Classification correction example

Suppose:

$$y_i = 1, \qquad F_0(x_i) = 0.$$

Then:

$$p_i = \frac{1}{1 + e^0} = 0.5.$$

The gradient and negative gradient are:

$$g_i = p_i - y_i = 0.5 - 1 = -0.5, \qquad r_i = -g_i = 0.5.$$

The model should increase the raw score because the true class is $1$ but the current probability is only $0.5$.

If a tree predicts correction $f_1(x_i) = 0.8$ and $\eta = 0.1$:

$$F_1(x_i) = 0 + 0.1(0.8) = 0.08.$$

The new probability is:

$$p_i = \frac{1}{1 + e^{-0.08}} \approx 0.5200.$$

The probability moved from $0.5$ to about $0.5200$, which is in the correct direction.

## 11. Decision Trees as Weak Learners

A decision tree partitions the feature space into terminal regions or leaves. If a tree has $J$ leaves, it can be written as:

$$f(x) = \sum_{j=1}^{J} w_j \mathbf{1}\{x \in R_j\},$$

where:

- $R_j$ is leaf region $j$.
- $w_j$ is the prediction value assigned to leaf $j$.

- $\mathbf{1}\{x \in R_j\}$ equals 1 if $x$ belongs to leaf $j$, and 0 otherwise.

#### Layman explanation

A tree asks a sequence of yes/no questions:

- Is income greater than a threshold?
- Is age below a threshold?
- Is transaction count above a threshold?

At the end, the observation lands in a leaf. Each leaf gives a number. In boosting, that number is usually a correction to the current prediction.

#### Small tree example

Suppose a tree uses one split:

$$\text{If area} < 1000 \text{ sq ft, go left; otherwise go right.}$$

Assume the leaf outputs are:

$$w_L = -30, \qquad w_R = 40.$$

Then:

$$f(x) = -30 \cdot \mathbf{1}\{\text{area} < 1000\} + 40 \cdot \mathbf{1}\{\text{area} \geq 1000\}.$$

If a house has area $900$, then $f(x) = -30$. If it has area $1200$, then $f(x) = 40$.

## 12. Regularization in Gradient Boosting

Gradient boosting can overfit if trees are too deep, the learning rate is too high, or the number of trees is too large. Regularization methods include:

- **Learning rate shrinkage**:

$$F_m(x) = F_{m-1}(x) + \eta f_m(x), \qquad 0 < \eta \leq 1.$$

- **Tree depth or leaf limits**, which restrict each tree's complexity.
- **Subsampling rows**, where each tree sees only part of the data.
- **Subsampling columns**, where each tree or split sees only part of the features.
- **Explicit penalties**, especially in XGBoost:

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

  where $T$ is the number of leaves, $\gamma$ penalizes adding leaves, and $\lambda$ penalizes large leaf weights.

#### Learning-rate example

Suppose:

$$F_0(x_i) = 100, \qquad f_1(x_i) = 50.$$

If $\eta = 1$:

$$F_1(x_i) = 100 + 1(50) = 150.$$

If $\eta = 0.1$:

$$F_1(x_i) = 100 + 0.1(50) = 105.$$

A smaller learning rate moves more cautiously. It usually requires more trees but often improves generalization.

# Part IV — XGBoost

## 13. What Is XGBoost?

**XGBoost** stands for **Extreme Gradient Boosting**. It is a scalable tree boosting system that extends classical gradient boosting with:

- second-order optimization using gradients and Hessians,
- explicit tree complexity regularization,
- sparsity-aware split finding,
- approximate split algorithms,
- column subsampling,
- efficient memory and parallel computation.

At boosting step $t$, the model is:

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

The regularized objective is:

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

A common tree penalty is:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2.$$

### Layman explanation

XGBoost is gradient boosting with stronger mathematics and stronger engineering. Classical gradient boosting says, “Use the direction of the error.” XGBoost says, “Use both the direction of the error and the curvature of the loss, and also punish trees that become unnecessarily complicated.”

Direction tells the model where to move. Curvature tells the model how cautious it should be.

### Model update example

Suppose:

$$\hat{y}^{(0)} = 10, \qquad f_1(x) = 2.$$

Then:

$$\hat{y}^{(1)} = 10 + 2 = 12.$$

If the next tree gives $f_2(x) = -0.5$, then:

$$\hat{y}^{(2)} = 12 - 0.5 = 11.5.$$

With a learning rate, many implementations update as:

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

## 14. XGBoost's Second-Order Taylor Approximation

XGBoost approximates the loss using a second-order Taylor expansion around the current prediction:

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

where:

$$g_i = \left[\frac{\partial L(y_i, \hat{y})}{\partial \hat{y}}\right]_{\hat{y}=\hat{y}_i^{(t-1)}},$$

and:

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

Ignoring constants independent of $f_t$, the approximate objective becomes:

$$\tilde{\mathcal{O}bj}^{(t)} = \sum_{i=1}^{n} \left[ g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t).$$

### Squared-error derivative example

Let:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2.$$

Then:

$$g = \frac{\partial L}{\partial \hat{y}} = \hat{y} - y, \qquad h = \frac{\partial^2 L}{\partial \hat{y}^2} = 1.$$

If $y = 10$ and $\hat{y} = 7$:

$$g = 7 - 10 = -3, \qquad h = 1.$$

The negative gradient is $3$, meaning the prediction should increase.

## 15. XGBoost Optimal Leaf Weight

Suppose a tree has $T$ leaves. Let $I_j$ be the set of training rows in leaf $j$. Define:

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

For a fixed tree structure, the objective contribution of leaf $j$ is:

$$G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2 + \gamma.$$

Differentiate with respect to $w_j$:

$$\frac{\partial}{\partial w_j}\left[G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2\right] = G_j + (H_j + \lambda) w_j.$$

Set to zero:

$$G_j + (H_j + \lambda) w_j = 0.$$

Therefore:

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

The optimal objective value for the tree is:

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

### Leaf-value example

Suppose one leaf contains three observations:

$$g_1 = -3, \qquad g_2 = -2, \qquad g_3 = 1,$$

and all Hessians are $1$:

$$h_1 = h_2 = h_3 = 1.$$

Then:

$$G = -3 - 2 + 1 = -4, \qquad H = 1 + 1 + 1 = 3.$$

With $\lambda = 1$:

$$w^* = -\frac{-4}{3 + 1} = 1.$$

This leaf adds $1$ to the current prediction for observations that land in it.

## 16. XGBoost Split Gain

A candidate split divides a parent node into left and right child nodes. Let:

$$G = G_L + G_R, \qquad H = H_L + H_R.$$

The split gain is:

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

XGBoost chooses splits with large positive gain.

#### Split-gain example

Suppose:

$$G_L = -4, \quad H_L = 3, \quad G_R = 2, \quad H_R = 2.$$

Then:

$$G = -4 + 2 = -2, \qquad H = 3 + 2 = 5.$$

Let $\lambda = 1$ and $\gamma = 0.1$.

$$\frac{G_L^2}{H_L+\lambda} = \frac{(-4)^2}{3+1} = 4,$$

$$\frac{G_R^2}{H_R+\lambda} = \frac{2^2}{2+1} = \frac{4}{3} \approx 1.3333,$$

$$\frac{G^2}{H+\lambda} = \frac{(-2)^2}{5+1} = \frac{4}{6} \approx 0.6667.$$

Therefore:

$$\text{Gain} = \frac{1}{2}(4 + 1.3333 - 0.6667) - 0.1 = 2.2333.$$

Because the gain is positive, the split is useful.

# Part V — LightGBM

## 17. What Is LightGBM?

**LightGBM** is a gradient boosting framework that uses tree-based learning algorithms and is designed for speed, memory efficiency, and scalability. Its major ideas include:

- histogram-based split finding,
- leaf-wise tree growth,
- Gradient-Based One-Side Sampling (GOSS),
- Exclusive Feature Bundling (EFB).

LightGBM asks:

> How can we make gradient boosting much faster without losing much accuracy?

It does this by:

- grouping continuous values into bins instead of checking every possible split,
- growing the most promising leaf first,
- keeping more rows with large errors and fewer rows with small errors,
- combining sparse features that rarely appear together.

### Why speed matters

Suppose a dataset has:

$$n = 1{,}000{,}000 \text{ rows}, \qquad d = 1{,}000 \text{ features}.$$

Checking every possible split for every feature is expensive. If each feature has many distinct values, split finding can dominate training time.

LightGBM reduces split search by replacing raw continuous feature values with bins, such as:

$$\text{age} \in \{0, 1, 2, \dots, 255\}.$$

Instead of checking hundreds of thousands of unique values, it checks at most $256$ bin thresholds per feature.

## 18. Histogram-Based Split Finding

LightGBM discretizes continuous features into bins:

$$x_i^{(k)} \mapsto b_i^{(k)} \in \{0, 1, \dots, B-1\},$$

where $B$ is the number of bins.

For each feature $k$ and bin $b$, LightGBM accumulates gradient and Hessian sums:

$$G_{k,b} = \sum_{i : b_i^{(k)} = b} g_i, \qquad H_{k,b} = \sum_{i : b_i^{(k)} = b} h_i.$$

For a threshold at bin $s$, the left child has:

$$G_L = \sum_{b=0}^{s} G_{k,b}, \qquad H_L = \sum_{b=0}^{s} H_{k,b}.$$

The right child has:

$$G_R = G - G_L, \qquad H_R = H - H_L.$$

The gain can use the same second-order form as other boosted tree systems:

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

### Histogram example

| Row | Feature value | Bin | Gradient $g_i$ | Hessian $h_i$ |
|---|---|---|---|---|
| 1 | 2.1 | 0 | -2 | 1 |
| 2 | 2.7 | 0 | -1 | 1 |
| 3 | 5.2 | 1 | 3 | 1 |
| 4 | 8.8 | 2 | 2 | 1 |

Bin statistics:

$$G_0 = -2 + (-1) = -3, \quad H_0 = 2,$$

$$G_1 = 3, \quad H_1 = 1,$$

$$G_2 = 2, \quad H_2 = 1.$$

A split at bin $0$ puts bin $0$ left and bins $1, 2$ right:

$$G_L = -3, \quad H_L = 2, \quad G_R = 3 + 2 = 5, \quad H_R = 1 + 1 = 2.$$

These statistics are inserted into the gain formula.

## 19. Leaf-Wise Tree Growth

Many tree algorithms grow trees **level-wise**, meaning all nodes at the same depth are split before moving deeper. LightGBM grows trees **leaf-wise**, also called **best-first** growth.

At each step, LightGBM chooses the leaf whose split produces the largest loss reduction:

$$j^* = \arg\max_j \Delta\mathcal{L}_j,$$

where $\Delta\mathcal{L}_j$ is the gain from splitting leaf $j$.

### Layman explanation

Level-wise tree growth is like building every room on floor 1 before starting floor 2. Leaf-wise growth asks:

> Which room expansion gives me the biggest benefit right now?

This can achieve lower loss with fewer splits. However, it can create deep, unbalanced trees, so LightGBM needs controls such as `num_leaves`, `max_depth`, and `min_data_in_leaf`.

### Choosing the best leaf

Suppose three leaves have possible split gains:

$$\Delta\mathcal{L}_1 = 0.3, \qquad \Delta\mathcal{L}_2 = 1.8, \qquad \Delta\mathcal{L}_3 = 0.6.$$

LightGBM chooses:

$$j^* = \arg\max(0.3, 1.8, 0.6) = 2.$$

It splits leaf $2$ first, not leaves $1$ and $3$ merely because they are at the same depth.

## 20. Gradient-Based One-Side Sampling (GOSS)

**Gradient-Based One-Side Sampling** keeps observations with large gradients and samples from observations with small gradients.

Let:

- $A$ be the set of top $a$ fraction of observations by absolute gradient.
- $B$ be a random sample of size $b$ fraction from the remaining observations.

For small-gradient sampled observations, LightGBM applies the multiplier:

$$\frac{1-a}{b}.$$

The adjusted gradient estimate is:

$$\tilde{G} = \sum_{i \in A} g_i + \frac{1-a}{b} \sum_{i \in B} g_i.$$

A similar adjusted sum can be used for split statistics.

### GOSS example

Suppose there are $100$ observations and:

$$a = 0.2, \qquad b = 0.3.$$

This means:

- keep the top $20\%$ by large gradient: $20$ observations,
- sample $30\%$ from the remaining $80$ observations: about $24$ observations.

The multiplier for sampled small-gradient observations is:

$$\frac{1-a}{b} = \frac{0.8}{0.3} \approx 2.6667.$$

If:

$$\sum_{i \in A} g_i = -5, \qquad \sum_{i \in B} g_i = 1.2,$$

then:

$$\tilde{G} = -5 + 2.6667(1.2) = -1.8.$$

## 21. Exclusive Feature Bundling (EFB)

**Exclusive Feature Bundling** combines sparse features that are rarely nonzero at the same time. If two features are mutually exclusive, they can be placed into one bundled feature without much information loss.

Two features $a$ and $b$ are exclusive if approximately:

$$x_i^{(a)} \neq 0 \Rightarrow x_i^{(b)} = 0,$$

and:

$$x_i^{(b)} \neq 0 \Rightarrow x_i^{(a)} = 0.$$

Bundling reduces the effective number of features:

$$d \to d_{\text{bundle}}, \qquad d_{\text{bundle}} < d.$$

### EFB example

| Row | Feature A | Feature B |
|---:|---:|---:|
| 1 | 1 | 0 |
| 2 | 0 | 1 |
| 3 | 0 | 0 |
| 4 | 1 | 0 |

Feature A and Feature B are never nonzero at the same time. They can be bundled. One possible representation is:

$$z_i = x_i^{(A)} + c \cdot x_i^{(B)},$$

where $c$ is an offset chosen so values from Feature A and Feature B do not collide.

If $c = 10$:

| Row | Bundled feature $z_i$ |
|---:|---:|
| 1 | 1 |
| 2 | 10 |
| 3 | 0 |
| 4 | 1 |

The model can still recover the split meaning because the value ranges are separated.

# Part VI — CatBoost

## 22. What Is CatBoost?

**CatBoost** stands for **Categorical Boosting**. It is a gradient boosting algorithm designed to handle categorical variables robustly and reduce target leakage.

Its major ideas include:

- native handling of categorical variables,
- ordered target statistics,
- ordered boosting,
- symmetric, or oblivious, trees.

Many real datasets contain categorical variables such as country, merchant, product type, customer segment, industry code, and branch name. Ordinary machine learning models need these categories converted into numbers. Bad conversion can leak the answer into the input features. CatBoost was designed to avoid that kind of leakage.

### Why naïve target encoding leaks

Suppose we have a categorical feature `City` and binary target `Default`:

| Row | City | Default |
|---|---|---|
| 1 | A | 1 |
| 2 | A | 0 |
| 3 | B | 1 |
| 4 | B | 1 |

A naïve target encoding for City A is:

$$\mathrm{TE}(A) = \frac{1+0}{2} = 0.5.$$

For City B:

$$\mathrm{TE}(B) = \frac{1+1}{2} = 1.$$

The problem is that row $3$ gets an encoded value of $1$ partly because its own target is $1$. For rare categories, this leakage can become severe.

## 23. Ordered Target Statistics

CatBoost uses permutations to compute target statistics using only earlier rows in the permutation.

Let $\sigma$ be a random permutation of the training rows. For row $i$, the ordered target statistic for category value $c$ can be written as:

$$\mathrm{OTS}_i(c) = \frac{\sum_{j:\sigma(j)<\sigma(i),\, x_j=c} y_j + ap}{\sum_{j:\sigma(j)<\sigma(i),\, x_j=c} 1 + a},$$

where:

- $a$ is a smoothing strength.
- $p$ is a prior, often the global target mean.
- only rows earlier than row $i$ in the permutation are used.

### Layman explanation

CatBoost pretends the data arrives in an order. When encoding a row, it only looks at previous rows, not the row's own answer. This is like calculating a batting average before the current game, not after including the current game's result.

### Ordered-encoding example

Suppose the permutation order is:

$$1 \to 2 \to 3 \to 4.$$

The data is:

| Row | City | Default |
|---|---|---|
| 1 | A | 1 |
| 2 | A | 0 |
| 3 | B | 1 |
| 4 | A | 1 |

The global prior is:

$$p = \frac{1+0+1+1}{4} = 0.75.$$

Let smoothing be $a = 1$.

For row $1$, City A has no previous City A rows:

$$\mathrm{OTS}_1(A) = \frac{0 + 1(0.75)}{0+1} = 0.75.$$

For row $2$, City A has one previous City A row with target $1$:

$$\mathrm{OTS}_2(A) = \frac{1 + 1(0.75)}{1+1} = 0.875.$$

For row $4$, City A has previous City A rows $1$ and $2$, with targets $1$ and $0$:

$$\mathrm{OTS}_4(A) = \frac{1+0+1(0.75)}{2+1} = \frac{1.75}{3} \approx 0.5833.$$

Notice row $4$'s own target is not included in its encoding.

## 24. Ordered Boosting

Standard boosting can suffer from **prediction shift** because the model's prediction for a training row may be influenced by target information from that same row through earlier fitted models.

CatBoost's ordered boosting uses permutations so that the residual or gradient for an observation is computed using a model trained only on observations earlier in the permutation.

Conceptually, for row $i$ under permutation $\sigma$, CatBoost aims to compute:

$$g_i = \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{<i}},$$

where $F_{<i}$ denotes a model constructed without using row $i$ or later rows in that permutation.

#### Layman explanation

CatBoost tries to make training behave more like real prediction. In real deployment, the model predicts new unseen rows; it does not get to use the target of the row it is predicting.

Ordered boosting imitates this: when estimating how wrong the model is on a row, it avoids letting that row's own target sneak into the model that produced the prediction.

#### Ordered-logic example

Suppose the permutation is:

$$A, B, C, D.$$

To compute the correction for row $C$, ordered boosting uses a model trained only from earlier rows:

$$\{A, B\}.$$

It does not use $\{C, D\}$. Therefore, row $C$ is treated more like a future unseen observation, reducing leakage and prediction shift.

## 25. Symmetric or Oblivious Trees

CatBoost commonly uses symmetric decision trees, also called **oblivious trees**. At each depth, the same split condition is applied across all nodes at that depth.

A depth-$D$ oblivious tree has at most:

$$2^D$$

leaves.

The leaf index can be computed from the binary outcomes of the $D$ split tests:

$$\ell(x) = \sum_{d=0}^{D-1} 2^d \mathbf{1}\{s_d(x) = 1\},$$

where $s_d(x)$ is the split decision at depth $d$.

#### Depth-2 example

Suppose a depth-2 oblivious tree uses:

$$s_0(x) = \mathbf{1}\{\text{Age} > 40\},$$

and:

$$s_1(x) = \mathbf{1}\{\text{Income} > 100000\}.$$

The leaf index is:

$$\ell(x) = 2^0 s_0(x) + 2^1 s_1(x).$$

If Age $> 40$ and Income $> 100000$, then:

$$s_0(x) = 1, \qquad s_1(x) = 1,$$

so:

$$\ell(x) = 1 + 2 = 3.$$

The observation lands in leaf $3$.

# Part VII — Choosing and Using the Algorithms

## 26. Conceptual Comparison

| Topic | Gradient Boosting | XGBoost | LightGBM | CatBoost |
|---|---|---|---|---|
| Core idea | Add weak learners to follow negative gradients | Gradient boosting with second-order regularized objective | Fast and memory-efficient GBDT | Boosting designed for categorical variables and reduced leakage |
| Main optimization | First-order functional gradient descent | Gradients plus Hessians | Histograms, GOSS, EFB, leaf-wise growth | Ordered target statistics and ordered boosting |
| Tree growth | Usually shallow trees | Depth-wise or configurable | Leaf-wise best-first | Symmetric/oblivious trees |
| Categorical features | Need preprocessing | Supports categorical features in newer versions but still often encoded | Supports categorical features, often with care | Native categorical handling is a core strength |
| Strength | Conceptual simplicity | Accuracy, regularization, scalability | Speed and memory efficiency | Minimal categorical preprocessing and strong defaults |
| Main risk | Overfitting if too many trees | Hyperparameter complexity | Overfitting with leaf-wise deep growth | Can be slower depending on settings and data |

## 27. Which Algorithm Should You Use?

Use **basic gradient boosting** when:

- teaching or prototyping,
- interpretability of fundamentals matters,
- dataset size is modest.

Use **XGBoost** when:

- strong predictive performance is needed,

- regularization control matters,
- sparse data or missing values are present,
- deployment maturity is important.

Use **LightGBM** when:

- the dataset is large,
- training speed is important,
- memory efficiency matters,
- there are many features or many rows.

Use **CatBoost** when:

- categorical variables are important,
- you want to avoid manual target-encoding leakage,
- you want strong out-of-the-box performance,
- preprocessing time should be minimized.

A simple rule of thumb:

- **Need to understand the concept?** Start with Gradient Boosting.
- **Need a powerful all-rounder?** Try XGBoost.
- **Need speed on large tabular data?** Try LightGBM.
- **Have many categorical fields?** Try CatBoost.

#### Choice example

Suppose you are building a default prediction model with these features:

- borrower income,
- age,
- loan amount,
- product type,
- branch code,
- industry code,
- customer segment.

If there are many categorical variables such as branch code and industry code, CatBoost is a natural first candidate. If the dataset has $10$ million rows and $1{,}000$ mostly numerical or sparse features, LightGBM is a natural first candidate. If you need carefully regularized performance and detailed control, XGBoost is a strong candidate.

## 28. Hyperparameters Explained Simply

### 28.1 Common hyperparameters

#### Number of trees.

$$M = \text{number of boosting rounds.}$$

This is the number of correction steps. Too few trees underfit; too many can overfit unless controlled by learning rate and early stopping.

**Learning rate.**

$$F_m(x) = F_{m-1}(x) + \eta f_m(x).$$

A small $\eta$ means careful learning. A large $\eta$ means aggressive learning.

**Maximum depth.** A tree with depth $D$ can have up to:

$$2^D$$

leaves in a full binary tree. Deeper trees can learn more complex interactions but can also memorize noise.

**Subsample.** If $q$ is the row subsampling rate, each boosting round trains on approximately:

$$qn$$

observations.

**Column sample.** If $c$ is the column subsampling rate, each tree or split considers approximately:

$$cd$$

features.

### 28.2 XGBoost-specific hyperparameters

- `lambda`: L2 regularization on leaf weights.
- `gamma`: minimum loss reduction needed to split.
- `min_child_weight`: minimum Hessian mass required in a child node.

A larger $\lambda$ shrinks leaf values:

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

If $\lambda$ increases, the denominator increases and $w_j^*$ decreases in magnitude.

### 28.3 LightGBM-specific hyperparameters

- `num_leaves`: maximum number of leaves.
- `min_data_in_leaf`: minimum number of rows per leaf.
- `max_bin`: number of histogram bins.
- `feature_fraction`: column sampling.
- `bagging_fraction`: row sampling.

For a full binary tree:

$$\text{num\_leaves} \leq 2^{\text{max\_depth}}.$$

For LightGBM, controlling `num_leaves` is crucial because leaf-wise growth can produce deep branches.

### 28.4 CatBoost-specific hyperparameters

- `iterations`: number of trees.
- `learning_rate`: step size.
- `depth`: depth of symmetric trees.
- `l2_leaf_reg`: L2 regularization.
- `cat_features`: categorical feature indices or names.

A depth-$D$ symmetric CatBoost tree has up to:

$$2^D$$

leaves.

## 29. Full Worked Mini-Example Across Algorithms

### 29.1 Dataset

| Row | Area | Age | True value $y$ |
|---|---|---|---|
| 1 | 800 | 20 | 100 |
| 2 | 1000 | 15 | 120 |
| 3 | 1200 | 10 | 150 |
| 4 | 1500 | 5 | 200 |

Initial prediction:

$$F_0 = \bar{y} = \frac{100 + 120 + 150 + 200}{4} = 142.5.$$

Residuals for squared-error gradient boosting are:

$$r_i = y_i - F_0(x_i).$$

Thus:

$$r_1 = -42.5, \quad r_2 = -22.5, \quad r_3 = 7.5, \quad r_4 = 57.5.$$

### 29.2 Gradient boosting step

A simple tree might split on:

$$\text{Area} < 1100.$$

Left rows: $1, 2$. Right rows: $3, 4$.

Left leaf residual mean:

$$w_L = \frac{-42.5 + (-22.5)}{2} = -32.5.$$

Right leaf residual mean:

$$w_R = \frac{7.5 + 57.5}{2} = 32.5.$$

If $\eta = 0.1$, predictions update as:

$$F_1 = 142.5 + 0.1(-32.5) = 139.25 \quad \text{for rows } 1, 2,$$

and:

$$F_1 = 142.5 + 0.1(32.5) = 145.75 \quad \text{for rows } 3, 4.$$

### 29.3 XGBoost step

For squared error:

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

At $F_0 = 142.5$:

$$g_1 = 42.5, \quad g_2 = 22.5, \quad g_3 = -7.5, \quad g_4 = -57.5.$$

For the same split, left node:

$$G_L = 42.5 + 22.5 = 65, \qquad H_L = 2.$$

Right node:

$$G_R = -7.5 - 57.5 = -65, \qquad H_R = 2.$$

Let $\lambda = 1$. Leaf weights are:

$$w_L^* = -\frac{65}{2+1} = -21.6667,$$

$$w_R^* = -\frac{-65}{2+1} = 21.6667.$$

XGBoost's regularization reduces the correction magnitude compared with ordinary residual means.

### 29.4 LightGBM step

LightGBM may first bin Area values:

| Area | Bin |
|---:|---:|
| 800 | 0 |
| 1000 | 0 |
| 1200 | 1 |
| 1500 | 2 |

It evaluates splits over bins rather than exact Area values. A split after bin $0$ produces the same grouping as `Area < 1100`:

- left: rows $1, 2$,
- right: rows $3, 4$.

The gain is computed using gradient and Hessian sums, while the histogram makes this faster.

### 29.5 CatBoost step

If there is also a categorical variable such as Region:

| Row | Region | True value $y$ |
|---:|---|---:|
| 1 | North | 100 |
| 2 | North | 120 |
| 3 | South | 150 |
| 4 | North | 200 |

A naïve encoding for North would be:

$$\frac{100 + 120 + 200}{3} = 140.$$

But for row $1$, this uses row $1$'s own target, which leaks information. CatBoost instead uses ordered statistics. If the permutation is $1, 2, 3, 4$, then row $2$'s North statistic can use row $1$, but row $1$ cannot use itself.

## 30. Common Mistakes and Safeguards

### 30.1 Data leakage

Data leakage occurs when training features contain information that would not be available at prediction time.

Example:

$$\text{Feature} = \text{average default rate of this exact customer including current outcome.}$$

This is invalid because it uses the answer to help predict the answer.

Safeguards include computing encodings and aggregations using cross-validation, time-aware splits, or ordered methods such as CatBoost's ordered statistics.

### 30.2 Improper validation

If data has time ordering, random train-test splits can be misleading. For time-dependent data, use:

$$\text{train period} < \text{validation period} < \text{test period}.$$

### 30.3 Too many trees without early stopping

Training loss usually decreases as trees are added, but validation loss may eventually increase. Let validation loss after $m$ trees be $\mathcal{L}_{\text{val}}^{(m)}$. Early stopping selects:

$$m^* = \arg\min_m \mathcal{L}_{\text{val}}^{(m)}.$$

### 30.4 Misinterpreting feature importance

Feature importance can be measured in different ways:

- split count,
- gain,
- permutation importance,
- SHAP values.

These do not always agree. A highly correlated feature may look less important if another correlated feature is selected first.

### 31. Final Practical Checklist

Before fitting a gradient boosting model, answer these questions:

- **What is the prediction target?** Regression, binary classification, multiclass classification, ranking, or survival?
- **What loss function is appropriate?** Squared error, logistic loss, cross-entropy, ranking loss, quantile loss, or another objective?
- **Are there categorical variables?** If yes, consider CatBoost or careful encoding.
- **Is the dataset large?** If yes, consider LightGBM or XGBoost with histogram methods.
- **Is there time dependence?** If yes, use time-aware validation.
- **Is leakage possible?** Check target encodings, aggregations, post-event variables, and future information.
- **Will the model be explained?** Plan feature importance, SHAP analysis, partial dependence, and validation diagnostics.
- **Will the model be monitored?** Track performance drift, feature drift, calibration, and stability.

## Part VIII — Historical Maturity Timeline

### 32. XGBoost, LightGBM, and CatBoost Maturity Timeline

Short answer: **XGBoost matured first, around 2016–2020; LightGBM matured around 2017–2021; CatBoost matured around 2018–2021/2022.**

However, “mature” depends on whether we mean:

- **Research maturity** — when the core algorithmic contribution was clearly published, validated, and recognized.
- **Industry adoption maturity** — when the tool became widely used in competitions, industry, and production workflows.
- **Software/API maturity** — when the package reached stable versioning, robust APIs, strong documentation, and production-ready releases.

| Library | Initial / Public Emergence | Research Maturity | Production / Stable Maturity | Practical Interpretation |
|---|---|---|---|---|
| **XGBoost** | Early-to-mid 2010s; major paper in **2016** | **2016**, after Chen & Guestrin's KDD paper formalized it as a scalable tree boosting system | **2020**, with **v1.0.0** marking semantic versioning, governance, and production ecosystem improvements | Mature earliest; by 2016 it was widely trusted in Kaggle and industry, and by 2020 it had stronger formal project maturity |
| **LightGBM** | **2016** original release period; paper in **2017** | **2017**, after the NeurIPS paper introduced GOSS and EFB | **2020–2021**, around v3.0.0/v3.3.0 when API, TreeSHAP, ranking, constraints, packaging, and sklearn support improved | Matured after XGBoost, mainly because it needed time to prove speed and large-scale advantages |
| **CatBoost** | Open-sourced in **July 2017** | **2018**, after the NeurIPS paper on ordered boosting and categorical features | **2021**, with **v1.0.0** described by the project as stable and production-ready | Matured last among the three, but became especially strong for categorical-heavy datasets |

### 32.1 XGBoost: mature by 2016, institutionally mature by 2020

XGBoost's major maturity point was **2016**, when the paper **"XGBoost: A Scalable Tree Boosting System"** was published and positioned XGBoost as a scalable, efficient, production-grade tree boosting system capable of handling very large datasets.

If maturity is defined as stable software governance and semantic versioning, then **February 19, 2020** is a stronger date. XGBoost **v1.0.0** was released as a major milestone, adopting Apache-style governance, semantic versioning, improved installation, better multi-core CPU scaling, Kubernetes support, and Dask integration.

Practical judgment:

- **Algorithm/research maturity: 2016**
- **Production/software maturity: 2020**
- **Overall mature-stage window: 2016–2020**

### 32.2 LightGBM: mature by 2017, operationally mature by 2020–2021

LightGBM emerged from Microsoft Research and was publicly associated with the **2017 NeurIPS paper**, which introduced:

- **Gradient-Based One-Side Sampling**, or **GOSS**
- **Exclusive Feature Bundling**, or **EFB**

The paper reported that LightGBM could speed up conventional Gradient Boosting Decision Tree training by up to over 20 times while maintaining almost the same accuracy.

From a software maturity perspective, LightGBM's mature stage is best placed around **2020–2021**. Version **v3.0.0** was released on **August 30, 2020**, and included production-relevant features such as improved Python/R support, interaction constraints, sparse TreeSHAP support, ranking enhancements,

monotone-constraint improvements, and better scikit-learn compatibility. LightGBM continued maturing through **v3.3.0 in October 2021**, after which it became a common default choice for large tabular datasets because of its speed, low memory usage, distributed capability, and GPU support.

Practical judgment:

- **Algorithm/research maturity: 2017**
- **Production/software maturity: 2020–2021**
- **Overall mature-stage window: 2017–2021**

### 32.3 CatBoost: mature by 2018, production-ready by 2021

CatBoost was open-sourced in **July 2017** by Yandex and was designed specifically to handle:

- categorical features,
- ordered boosting,
- GPU training,
- visualization,
- model-analysis tooling.

Its research maturity point was **2018**, when the NeurIPS paper **"CatBoost: Unbiased Boosting with Categorical Features"** formally introduced and analyzed ordered boosting and categorical-feature handling to reduce prediction shift and target leakage.

Its clearest software maturity point was **v1.0.0**, where the project stated that CatBoost was considered stable and production-ready and already used in many companies and individual projects.

Practical judgment:

- **Algorithm/research maturity: 2018**
- **Production/software maturity: 2021**
- **Overall mature-stage window: 2018–2021**

### 32.4 Final ranking by maturity timing

1. **XGBoost** — matured first, around **2016**, with software/governance maturity by **2020**.
2. **LightGBM** — matured next, around **2017**, with operational maturity around **2020–2021**.
3. **CatBoost** — matured after that, around **2018**, with explicit production-ready maturity around **2021**.

## Part IX — Mathematical Summary and Glossary

## 33. Mathematical Summary

### 33.1 Generic gradient boosting

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \eta f_m(x).$$

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}.$$

$$f_m \approx \arg\min_f \sum_{i=1}^{n} (r_{im} - f(x_i))^2.$$

### 33.2 Logistic classification

$$p_i = \frac{1}{1 + e^{-F(x_i)}}.$$

$$L(y_i, F(x_i)) = -y_i \log(p_i) - (1 - y_i)\log(1 - p_i).$$

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

### 33.3 XGBoost objective

$$\mathcal{O}bj^{(t)} = \sum_{i=1}^{n} L\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t).$$

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2.$$

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

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

### 33.4 LightGBM histogram

$$x_i^{(k)} \mapsto b_i^{(k)} \in \{0, 1, \dots, B-1\}.$$

$$G_{k,b} = \sum_{i : b_i^{(k)} = b} g_i, \qquad H_{k,b} = \sum_{i : b_i^{(k)} = b} h_i.$$

### 33.5 CatBoost ordered target statistic

$$\text{OTS}_i(c) = \frac{\sum_{j : \sigma(j) < \sigma(i),\, x_j = c} y_j + a p}{\sum_{j : \sigma(j) < \sigma(i),\, x_j = c} 1 + a}.$$

## 34. Glossary and Interpretation of Mathematical Formulae

This section collects the main formulae, symbols, and interpretations used throughout the document.

### 34.1 Additive boosting model

Formula:

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \eta f_m(x).$$

Symbols:

- $x$: input observation or feature vector.
- $F_M(x)$: final model prediction after $M$ boosting rounds.
- $F_0(x)$: initial prediction before any trees are added.
- $M$: total number of boosting iterations or trees.
- $m$: boosting-round index.
- $\eta$: learning rate or shrinkage.
- $f_m(x)$: weak learner added at round $m$, usually a decision tree.
- $\sum$: sum over boosting rounds.

Example: if $F_0(x) = 100$, $f_1(x) = 20$, $f_2(x) = -5$, and $\eta = 0.1$, then:

$$F_2(x) = 100 + 0.1(20) + 0.1(-5) = 101.5.$$

### 34.2 Empirical loss function

Formula:

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

Symbols:

- $\mathcal{L}(F)$: total loss of the model.
- $L(y_i, F(x_i))$: loss for observation $i$.
- $y_i$: true target value.
- $F(x_i)$: model prediction.
- $n$: number of observations.
- $i$: observation index.

If three observation-level losses are $4, 9, 1$, then total loss is $14$.

### 34.3 Gradient descent parameter update

Formula:

$$\theta_m = \theta_{m-1} - \eta \nabla_\theta \mathcal{L}(\theta_{m-1}).$$

The gradient points uphill in loss. Subtracting it moves downhill. If $\theta_{m-1} = 10$, $\nabla_\theta \mathcal{L} = 3$, and $\eta = 0.1$, then:

$$\theta_m = 10 - 0.1(3) = 9.7.$$

### 34.4 Pseudo-residual or negative gradient

Formula:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}.$$

The pseudo-residual tells the next tree what needs to be fixed. For squared error, it becomes the ordinary residual:

$$r_i = y_i - F(x_i).$$

If $y = 100$ and $F = 80$, then $r = 20$, so the model should increase its prediction.

### 34.5 Squared-error loss and residual

Loss:

$$L(y_i, F(x_i)) = \frac{1}{2}(y_i - F(x_i))^2.$$

Residual:

$$r_i = y_i - F(x_i).$$

The factor $\frac{1}{2}$ does not change the optimization target; it simplifies derivatives. If $y_i = 100$ and $F(x_i) = 90$, then:

$$L = \frac{1}{2}(10)^2 = 50.$$

### 34.6 Logistic sigmoid, loss, gradient, and Hessian

Sigmoid:

$$p_i = \sigma(F(x_i)) = \frac{1}{1 + e^{-F(x_i)}}.$$

Binary logistic loss:

$$L(y_i, F(x_i)) = -y_i \log(p_i) - (1 - y_i)\log(1 - p_i).$$

Gradient:

$$g_i = \frac{\partial L}{\partial F(x_i)} = p_i - y_i.$$

Hessian:

$$h_i = p_i(1 - p_i).$$

The sigmoid maps any real-valued score to a probability between $0$ and $1$. The gradient tells whether the raw score is too high or too low. The Hessian measures curvature and is largest near $p_i = 0.5$.

Examples:

- If $F(x_i) = 0$, then $p_i = 0.5$.
- If $y_i = 1$ and $p_i = 0.8$, then $L = -\log(0.8) \approx 0.2231$.
- If $y_i = 1$ and $p_i = 0.3$, then $g_i = -0.7$ and the negative gradient is $0.7$.

- If $p_i = 0.5$, then $h_i = 0.25$; if $p_i = 0.9$, then $h_i = 0.09$.

### 34.7 Decision tree as leaf regions

Formula:

$$f(x) = \sum_{j=1}^{J} w_j \mathbf{1}\{x \in R_j\}.$$

A decision tree partitions feature space into $J$ disjoint leaf regions. Each observation belongs to one leaf region, and the tree returns that leaf's weight.

### 34.8 XGBoost objective, penalty, Taylor approximation, leaf weight, and gain

Objective:

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

Penalty:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2.$$

Taylor approximation:

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

Gradient and Hessian definitions:

$$g_i = \left[\frac{\partial L(y_i, \hat{y})}{\partial \hat{y}}\right]_{\hat{y}=\hat{y}_i^{(t-1)}}, \qquad h_i = \left[\frac{\partial^2 L(y_i, \hat{y})}{\partial \hat{y}^2}\right]_{\hat{y}=\hat{y}_i^{(t-1)}}.$$

Optimal leaf weight:

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

Split gain:

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

Here $G_j$ and $H_j$ are sums of gradients and Hessians in leaf $j$. $\gamma$ penalizes additional leaves or splits, and $\lambda$ shrinks leaf weights.

### 34.9 LightGBM histogram, leaf-wise growth, GOSS, and EFB

Histogram binning:

$$x_i^{(k)} \mapsto b_i^{(k)} \in \{0, 1, \ldots, B-1\}.$$

Gradient and Hessian bin sums:

$$G_{k,b} = \sum_{i:b_i^{(k)}=b} g_i, \qquad H_{k,b} = \sum_{i:b_i^{(k)}=b} h_i.$$

Leaf-wise rule:

$$j^* = \arg\max_j \Delta \mathcal{L}_j.$$

GOSS adjusted gradient estimate:

$$\tilde{G} = \sum_{i \in A} g_i + \frac{1-a}{b} \sum_{i \in B} g_i.$$

Exclusive feature condition:

$$x_i^{(a)} \neq 0 \Rightarrow x_i^{(b)} = 0, \qquad x_i^{(b)} \neq 0 \Rightarrow x_i^{(a)} = 0.$$

Bundled feature representation:

$$z_i = x_i^{(A)} + c \cdot x_i^{(B)}.$$

### 34.10 CatBoost ordered statistics, ordered boosting, and symmetric trees

Ordered target statistic:

$$\mathrm{OTS}_i(c) = \frac{\sum_{j:\sigma(j)<\sigma(i),\, x_j=c} y_j + ap}{\sum_{j:\sigma(j)<\sigma(i),\, x_j=c} 1 + a}.$$

Ordered boosting gradient:

$$g_i = \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F=F_{<i}}.$$

Symmetric tree leaf count:

$$2^D.$$

Leaf index:

$$\ell(x) = \sum_{d=0}^{D-1} 2^d \mathbf{1}\{s_d(x) = 1\}.$$

### 34.11 Early stopping, row sampling, and column sampling

Early stopping:

$$m^* = \arg\min_m \mathcal{L}_{\text{val}}^{(m)}.$$

Row subsampling count:

$$qn.$$

Column subsampling count:

$$cd.$$

Full binary tree leaf limit:

$$\text{num\_leaves} \leq 2^{\text{max\_depth}}.$$

Initial prediction for squared-error regression:

$$F_0(x) = \bar{y}, \qquad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

## 34.12 Master glossary of common symbols

### Dataset and prediction symbols

- $x$: input observation or feature vector.
- $x_i$: input features for observation $i$.
- $x_i^{(k)}$: value of feature $k$ for observation $i$.
- $y_i$: true target value for observation $i$.
- $\hat{y}_i$: predicted value for observation $i$.
- $F(x_i)$: model prediction for observation $i$.
- $F_m$: model after $m$ boosting rounds.
- $F_0$: initial prediction.
- $f_m$: weak learner added at round $m$.

### Indexing symbols

- $i$: observation index.
- $j$: leaf index or generic index.
- $m$: boosting round index.
- $t$: boosting round index, commonly used in XGBoost notation.
- $k$: feature index.
- $d$: tree-depth index.
- $n$: number of observations.
- $M$: total number of boosting rounds.
- $D$: tree depth.
- $T$: number of leaves in XGBoost notation.
- $J$: number of leaves or terminal regions.

### Loss and optimization symbols

- $L$: individual loss.
- $\mathcal{L}$: total loss.
- $\mathcal{O}bj^{(t)}$: XGBoost objective at round $t$.
- $\Omega(f_t)$: regularization penalty for tree $t$.
- $\eta$: learning rate.
- $\lambda$: L2 regularization strength.
- $\gamma$: split or leaf complexity penalty.
- $\arg\max$: input value that maximizes a function.
- $\arg\min$: input value that minimizes a function.

### Gradient and Hessian symbols

- $g_i$: gradient for observation $i$.

- $h_i$: Hessian for observation $i$.
- $G_j$: sum of gradients in leaf $j$.
- $H_j$: sum of Hessians in leaf $j$.
- $G_L, H_L$: gradient and Hessian sums in the left child.
- $G_R, H_R$: gradient and Hessian sums in the right child.

#### Tree symbols

- $R_j$: leaf region $j$.
- $w_j$: weight or value of leaf $j$.
- $w_j^*$: optimal leaf weight.
- $\mathbf{1}\{\cdot\}$: indicator function.
- $\ell(x)$: leaf index for observation $x$.
- $s_d(x)$: split decision at depth $d$.

#### LightGBM symbols

- $B$: number of histogram bins.
- $b_i^{(k)}$: bin index for feature $k$ of observation $i$.
- $G_{k,b}$: gradient sum for feature $k$ and bin $b$.
- $H_{k,b}$: Hessian sum for feature $k$ and bin $b$.
- $A$: large-gradient sample set in GOSS.
- $B$ in GOSS context: small-gradient sample set. Context distinguishes it from number of bins.
- $a$: fraction of large-gradient observations retained.
- $b$: fraction sampled from small-gradient observations.

#### CatBoost symbols

- $\sigma$: random permutation of observations.
- $\sigma(i)$: position of observation $i$ in the permutation.
- $\mathrm{OTS}_i(c)$: ordered target statistic for row $i$ and category $c$.
- $c$: category value.
- $a$: smoothing parameter in ordered target statistics.
- $p$: prior value, often global target mean.
- $F_{<i}$: model trained only on observations before row $i$ in a permutation.

### 34.13 Final interpretation

All the formulae in gradient boosting, XGBoost, LightGBM, and CatBoost revolve around one core idea:

Learn from the remaining error, add a controlled correction, and repeat.

Gradient Boosting focuses on the negative gradient. XGBoost improves this with second-order information and explicit regularization. LightGBM improves computational efficiency using histograms, sampling, and feature bundling. CatBoost improves categorical-feature handling and reduces leakage using ordered methods.

In plain English:

> Boosting is a disciplined process of repeatedly asking: what mistakes remain, how should I correct them, and how do I avoid correcting them too aggressively?

# Appendix A — Vectorization, BLAS, Distributed Computing, and GPUs

This appendix builds on the scalar-gradient and scalar-Hessian explanation. The key idea is:

> In ordinary regression and binary classification boosting, each row has a scalar gradient and scalar Hessian value. Across a large dataset, these scalar values become large vectors that can be processed efficiently using vectorized CPU operations, BLAS-style numerical routines, GPU parallelism, and distributed computing.

## A.1 Scalar gradient and scalar Hessian per row

The Hessian looks like a single number because the model is usually predicting one number per row:

- in regression, one predicted value;
- in binary classification, one raw score that becomes a probability.

Thus:

$$\text{scalar prediction} \Rightarrow \text{scalar gradient and scalar Hessian value.}$$

For one row $i$:

$$g_i = \frac{\partial L_i}{\partial F(x_i)}, \qquad h_i = \frac{\partial^2 L_i}{\partial F(x_i)^2}.$$

If the model predicts several numbers at once, such as one score per class in multiclass classification, then the gradient becomes a vector and the Hessian becomes a matrix.

## A.2 From one row to the whole dataset

For a dataset with $n$ rows, each row has its own gradient:

$$g_1, g_2, \ldots, g_n,$$

and Hessian:

$$h_1, h_2, \ldots, h_n.$$

These can be collected into vectors:

$$g = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}, \qquad h = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{bmatrix}.$$

For binary classification:

$$g = p - y,$$

where:

$$p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix}, \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

and:

$$h = p \odot (1 - p),$$

where $\odot$ means element-wise multiplication.

Vectorized thinking says:

> Calculate the errors for all rows at the same time.

Modern CPUs and GPUs are designed to perform many similar numerical operations in parallel.

## A.3 Why this matters for boosting

For a leaf $j$, boosted-tree algorithms compute:

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

where $I_j$ is the set of rows falling into leaf $j$.

The optimal leaf weight is:

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

Training repeatedly performs operations such as:

- calculating gradients,
- calculating Hessians,
- grouping rows by tree split,
- summing gradients and Hessians in each group,
- computing gain,
- choosing the best split.

These operations repeat across many rows, features, candidate split points, and trees. Efficient vectorized computation is therefore crucial.

## A.4 BLAS-style matrix and vector operations

BLAS stands for **Basic Linear Algebra Subprograms**. BLAS-style operations are highly optimized routines for vector and matrix computation, including:

- vector addition: $z = x + y$,
- scalar-vector multiplication: $z = \alpha x$,
- dot product: $s = x^\top y$,
- matrix-vector multiplication: $z = X\beta$,
- matrix-matrix multiplication: $C = AB$.

Boosted trees are not purely matrix-multiplication models like neural networks, but they still benefit from BLAS-style thinking because training involves large-scale vector updates, reductions, sorting, histogram construction, and aggregation.

For boosting, one important vectorized update is:

$$F^{(t)} = F^{(t-1)} + \eta f_t(X),$$

where $F^{(t)}$ is the vector of current predictions for all rows, $F^{(t-1)}$ is the previous prediction vector, $f_t(X)$ is the vector of predictions from the new tree, and $\eta$ is the learning rate.

## A.5 Reduction operations

Boosting relies heavily on **reduction operations**, which compress many numbers into one number:

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

If a node contains one million rows, a parallel implementation can split the work into chunks:

$$G_j = \left(\sum_{i \in A_1} g_i\right) + \left(\sum_{i \in A_2} g_i\right) + \cdots + \left(\sum_{i \in A_K} g_i\right).$$

Each chunk can be computed independently. This is the basic idea behind parallel reduction.

## A.6 Histogram-based boosting and massive parallelism

LightGBM and modern XGBoost implementations often use histogram-based split finding:

$$x_i^{(k)} \mapsto b_i^{(k)}.$$

For each feature $k$ and bin $b$:

$$G_{k,b} = \sum_{i : b_i^{(k)} = b} g_i, \qquad H_{k,b} = \sum_{i : b_i^{(k)} = b} h_i.$$

This is highly parallelizable because different features, bins, and data partitions can be processed at the same time.

## A.7 Why GPUs help

GPUs are designed for massive parallel computation. A CPU usually has fewer powerful cores, while a GPU has many smaller cores designed to perform the same type of operation across many data points.

Boosting can use GPUs effectively for:

- computing gradients for many rows,
- computing Hessians for many rows,
- building histograms,
- scanning split candidates,
- summing gradient and Hessian statistics,
- updating predictions,
- evaluating many features in parallel.

Gradient and Hessian vector operations such as:

$$g = p - y, \qquad h = p \odot (1 - p)$$

are naturally suitable for GPU parallelism.

## A.8 Why distributed computing helps

Distributed computing spreads computation across multiple machines. This matters when the dataset is too large for one machine or training needs to be faster.

Suppose the dataset is split across $K$ workers:

$$D = D_1 \cup D_2 \cup \cdots \cup D_K.$$

Each worker computes local sums:

$$G_j^{(k)} = \sum_{i \in I_j \cap D_k} g_i, \qquad H_j^{(k)} = \sum_{i \in I_j \cap D_k} h_i.$$

Then global sums are aggregated:

$$G_j = \sum_{k=1}^{K} G_j^{(k)}, \qquad H_j = \sum_{k=1}^{K} H_j^{(k)}.$$

Each machine works on part of the dataset, then the partial totals are combined.

## A.9 Why scalar Hessians still enable huge speedups

At first, scalar Hessians may look too simple to justify advanced computing. Their simplicity is exactly what makes them fast. For each row:

$$h_i \in \mathbb{R}.$$

For all rows:

$$h = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{bmatrix}, \qquad g = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}.$$

Large vectors are ideal for vectorized computation. The algorithm does not need expensive full matrix Hessians for ordinary binary classification or regression. Instead, it uses cheap scalar second derivatives per row, then aggregates them efficiently.

## A.10 Connection to split gain

Recall:

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

The formula only needs aggregated values:

- $G_L$,
- $H_L$,
- $G_R$,
- $H_R$,
- $G$,
- $H$.

The algorithm does not need a full derivative structure for every row when evaluating a split. It summarizes rows into totals, then compares totals.

## A.11 Difference from full matrix methods

Some machine learning methods require solving large matrix systems, such as:

$$(X^\top X)^{-1} X^\top y,$$

or computing large Hessian matrices:

$$H \in \mathbb{R}^{p \times p},$$

where $p$ is the number of model parameters.

Boosted trees often avoid this full dense matrix operation. Instead, they rely on:

- scalar gradients per row,
- scalar Hessians per row,
- histogram aggregation,
- split gain summaries,
- parallel reductions.

This structure is suitable for vectorization, cache-efficient CPU code, SIMD instructions, GPU kernels, distributed aggregation, and memory-efficient histogram algorithms.

## A.12 Final appendix summary

The Hessian may look like "just one number" per row, but across a large dataset, those numbers become a huge vector of curvature information.

In simple terms:

$$\text{one row} \Rightarrow \text{scalar gradient and scalar Hessian},$$

but:

$$\text{millions of rows} \Rightarrow \text{large gradient and Hessian vectors}.$$

Large vectors can be processed quickly using:

- vectorized CPU operations,
- BLAS-style numerical routines,
- SIMD instructions,
- GPU parallelism,
- distributed computing,
- histogram-based aggregation.

The math is scalar per row, but the computation is massively parallel across rows, features, bins, trees, and machines. That is why XGBoost, LightGBM, and CatBoost can achieve large speedups on modern hardware.

# Bibliography and Source Notes

The following references are useful for deeper study:


- Jerome H. Friedman, **"Greedy Function Approximation: A Gradient Boosting Machine"**, *The Annals of Statistics*, 2001. DOI: `10.1214/aos/1013203451`.
  URL: https://projecteuclid.org/journals/annals-of-statistics/volume-29/issue-5/Greedy-function-approximation-A-gradient-boosting-machine/10.1214/aos/1013203451.full
- Tianqi Chen and Carlos Guestrin, **"XGBoost: A Scalable Tree Boosting System"**, KDD 2016.
  URL: https://arxiv.org/abs/1603.02754
- XGBoost documentation.
  URL: https://xgboost.readthedocs.io/
- Guolin Ke et al., **"LightGBM: A Highly Efficient Gradient Boosting Decision Tree"**, NeurIPS 2017.
  URL: https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree
- LightGBM documentation.
  URL: https://lightgbm.readthedocs.io/en/latest/
- Liudmila Prokhorenkova et al., **"CatBoost: Unbiased Boosting with Categorical Features"**, NeurIPS 2018.
  URL: https://proceedings.neurips.cc/paper/2018/hash/14491b756b3a51daac41c24863285549-

Abstract.html
- CatBoost documentation on categorical features.
  URL: https://catboost.ai/docs/en/features/categorical-features

## Closing Summary

Gradient boosting builds a strong model by adding many small correction models. XGBoost strengthens this idea with second-order optimization and explicit regularization. LightGBM makes it faster and more memory efficient through histograms, leaf-wise growth, GOSS, and EFB. CatBoost focuses on categorical variables and leakage-reducing ordered methods.

The shared foundation is:

$$\text{strong model} = \text{initial guess} + \text{many small corrections}.$$

The differences are mainly about how each algorithm chooses, regularizes, accelerates, and safely computes those corrections.