Gradient Boosting: Complete Guide

Including XGBoost, LightGBM & CatBoost

Introduction

Gradient Boosting is one of the most popular machine learning algorithms. This is known as Data Science's silver bullet because of its outstanding performance in various real world applications.

While the gradient boost algorithm looks complicated because it was designed to be configured in many ways, the core concept behind this algorithm is very straightforward.

What is Gradient Boosting? Gradient Boosting is a machine learning technique that builds an ensemble of weak learners, typically decision trees, in a stage-wise fashion to minimize a loss function. The original algorithm is often referred to as GBM (Gradient Boosted Machine).

In practice, we rarely use the basic GBM implementation directly; instead, we rely on highly optimized libraries such as XGBoost, LightGBM, or CatBoost, which implement gradient boosting with additional engineering improvements for speed, scalability, and accuracy.

Prerequisites: This presentation assumes familiarity with classification and regression trees, AdaBoost, and the regularization.

AdaBoost vs Gradient Boost Comparison

Aspect	**AdaBoost**	**Gradient Boost**
Initial Prediction	Starts with a stump (very short tree)	Starts with a single leaf (initial guess)
Tree Size	Usually stumps (2 leaves)	Larger trees (4-32 leaves typically)
Scaling	Each tree scaled by different amounts based on performance	All trees scaled by the same learning rate
Tree Building	Based on errors from previous stumps	Based on residuals from previous trees

Part 1: Gradient Boost for Regression

Training Data Example

We'll use this dataset to predict weight based on height, favorite color, and gender:

Person	Height (cm)	Favorite Color	Gender	Weight (kg)
1	180	Blue	Male	88
2	170	Red	Female	76
3	175	Green	Male	80
4	160	Blue	Female	70
5	165	Red	Male	65
6	155	Green	Female	48

Step 1: Initial Prediction

For regression, the first guess is the average of all target values:

$$\text{Initial Prediction} = \frac{88 + 76 + 80 + 70 + 65 + 48}{6} = 71.2 \text{ kg}$$

Initial Leaf

71.2 kg

Step 2: Calculate Pseudo-Residuals

Pseudo-residuals are the differences between observed and predicted values:

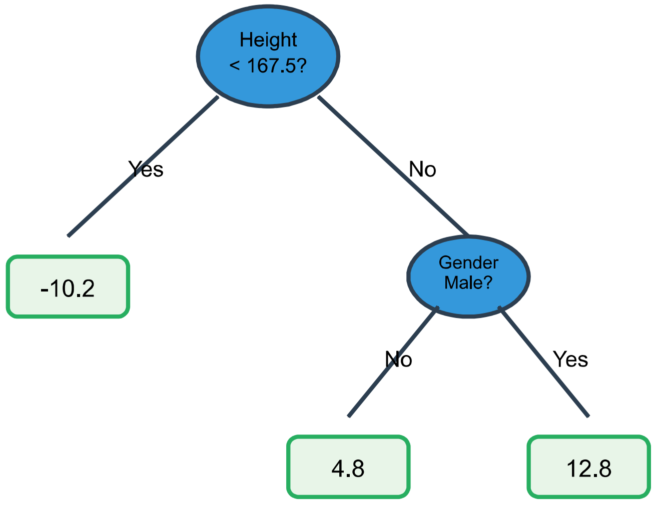
$$\text{Pseudo-Residual} = \text{Observed} - \text{Predicted}$$

Person	Observed Weight	Predicted Weight	Pseudo-Residual
1	88	71.2	16.8
2	76	71.2	4.8
3	80	71.2	8.8
4	70	71.2	-1.2
5	65	71.2	-6.2

Person	Observed Weight	Predicted Weight	Pseudo-Residual
6	48	71.2	-23.2

Step 3: Build First Tree

Build a tree to predict the pseudo-residuals using the input features:



Note: In this example, we limit trees to 4 leaves maximum. In practice, 8–32 leaves are common.

Step 4: Apply Learning Rate

To prevent overfitting, we scale the tree's contribution with a learning rate (typically 0.1):

$$\text{New Prediction} = \text{Previous Prediction} + (\text{Learning Rate} \times \text{Tree Output})$$

$$\text{Learning Rate} = 0.1$$

Insight: Empirical evidence shows that taking lots of small steps in the right direction results in better predictions with testing data (lower variance).

For Person 1: $71.2 + (0.1 \times 12.8) = 71.2 + 1.28 = 72.48$ kg

Step 5: Iterate

Continue building trees based on new residuals until:

- Maximum number of trees is reached, OR
- Additional trees don't significantly reduce residuals

Gradient Boost Regression Algorithm:

1. Start with initial prediction (average)
2. Calculate pseudo-residuals
3. Build tree to predict residuals
4. Scale tree output by learning rate
5. Update predictions
6. Repeat steps 2-5 until convergence

Final Prediction

For new data, combine all trees:

$$\text{Final Prediction} = \text{Initial Leaf} + \alpha \sum_{i=1}^{n} \text{Tree}_i(\mathbf{x})$$

where α is the learning rate

Part 2: Gradient Boost for Classification

Training Data Example

We'll use this dataset to classify whether someone will purchase a product based on their profile:

Customer	Age	Income ($1000s)	Has Credit Card	Will Purchase
1	25	45	Yes	Yes
2	35	55	No	No
3	45	75	Yes	Yes
4	28	35	No	No
5	40	80	Yes	Yes
6	32	60	Yes	Yes

Step 1: Initial Prediction (Log-Odds)

For classification, the initial prediction is the log-odds (logistic regression equivalent of average):

$$\text{Initial Prediction} = \log\left(\frac{\#\text{ who will purchase}}{\#\text{ who won't purchase}}\right) = \log\left(\frac{4}{2}\right) = 0.7$$

Log-Odds

0.7

Step 2: Convert to Probability

Use the logistic function to convert log-odds to probability:

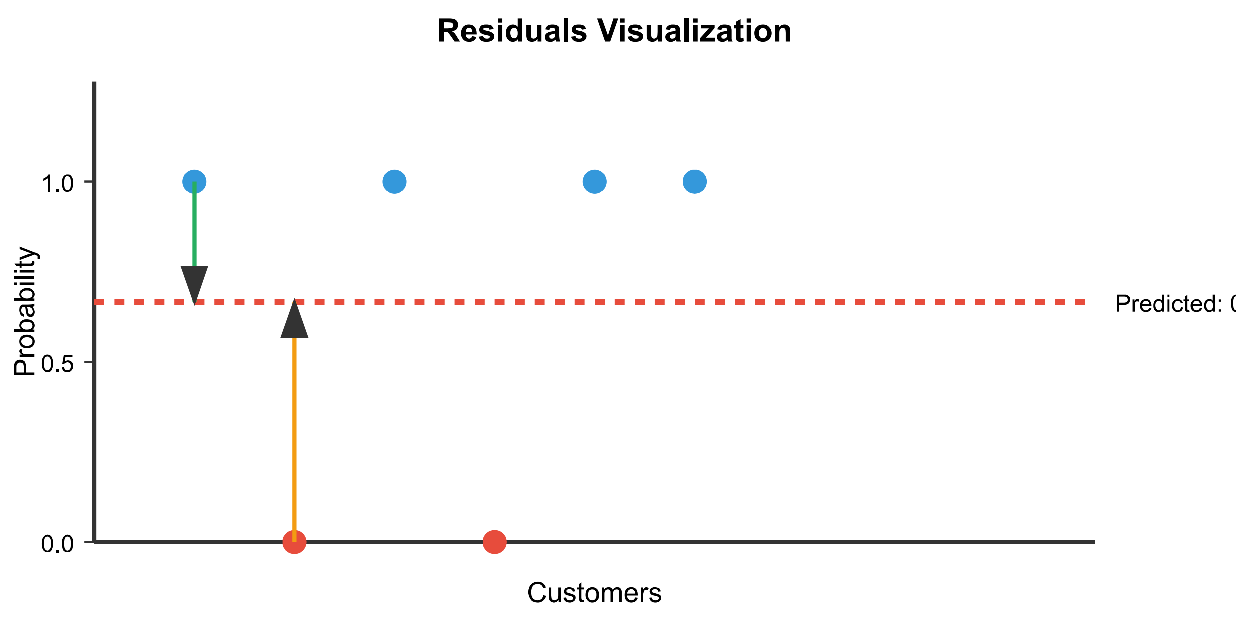
$$P(\text{Will Purchase}) = \frac{e^{\text{log-odds}}}{1 + e^{\text{log-odds}}} = \frac{e^{0.7}}{1 + e^{0.7}} = 0.67$$

Since 0.67 > 0.5, we classify everyone as "Will Purchase" initially.

Step 3: Calculate Pseudo-Residuals

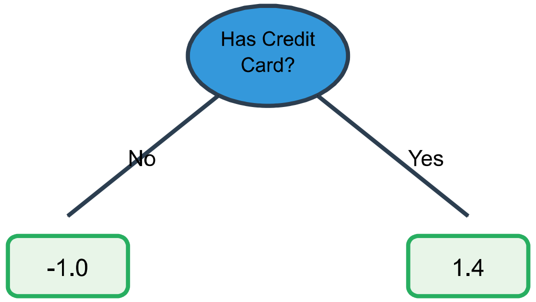
For classification, residuals are differences between observed and predicted probabilities:

Customer	Observed (0/1)	Predicted Prob	Pseudo-Residual
1	1	0.67	0.33
2	0	0.67	-0.67
3	1	0.67	0.33
4	0	0.67	-0.67
5	1	0.67	0.33
6	1	0.67	0.33



Step 4: Build Tree and Calculate Output Values

Build a tree to predict residuals, then calculate output values using a special formula:



Output Value Formula for Classification:

$$\text{Output Value} = \frac{\sum \text{Residuals in Leaf}}{\sum P_{prev}(1 - P_{prev})}$$

where P_{prev} is the previous predicted probability

Example calculation for right leaf:

Residuals: 0.33, 0.33, 0.33, 0.33 (4 customers with credit cards)

Previous probability for all: 0.67

$$\text{Output} = \frac{0.33 + 0.33 + 0.33 + 0.33}{4 \times 0.67 \times (1 - 0.67)} = \frac{1.32}{4 \times 0.67 \times 0.33} = \frac{1.32}{0.88} = 1.4$$

Step 5: Update Predictions

Combine the initial leaf with the scaled tree output (using learning rate α = 0.8):

Note: Here we have used much larger value of learning rate (α) for illustration purpose. Usually much smaller value like 0.1 is used for learning rate.

$$\text{New Log-Odds} = \text{Previous Log-Odds} + (\alpha \times \text{Tree Output})$$

For a customer with a credit card:

$$\text{New Log-Odds} = 0.7 + (0.8 \times 1.4) = 0.7 + 1.12 = 1.82$$

Convert back to probability:

$$P = \frac{e^{1.82}}{1 + e^{1.82}} = 0.86$$

Step 6: Iterate

Continue the process:

1. Calculate new residuals
2. Build new tree
3. Calculate output values
4. Update predictions
5. Repeat until convergence

Final Classification

For new data, combine all components:

$$\boxed{\text{Final Log-Odds} = \text{Initial Leaf} + \alpha \sum_{i=1}^{n} \text{Tree}_i(\mathbf{x})}$$

Convert to probability and apply threshold (typically 0.5):

$$P = \frac{e^{\text{log-odds}}}{1 + e^{\text{log-odds}}}$$

If $P > 0.5$: Classify as "Will Purchase"

If $P \leq 0.5$: Classify as "Won't Purchase"

Key Takeaways

Gradient Boost Summary

- **Regression:** Start with average, build trees on residuals, scale by learning rate
- **Classification:** Start with log-odds, build trees on probability residuals, use special output formula
- **Learning Rate:** Small values (0.1 typical) take many small steps for better generalization
- **Tree Size:** Typically 8-32 leaves, larger than AdaBoost stumps
- **Stopping:** When max trees reached or residuals don't improve significantly

Core Insight: Taking lots of small steps in the right direction (via learning rate) results in better predictions on test data, reducing overfitting and variance.

Mathematical Foundation

General Gradient Boosting Framework

Objective: Minimize loss function $L(y, F(\mathbf{x}))$

Regression (Squared Error):

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

Classification (Logistic Loss):

- For labels $y \in \{-1, +1\}$:

$$L(y, F) = \log(1 + e^{-yF})$$

- For labels $y \in \{0, 1\}$:

$$L(y, F) = -\left[y \log(p) + (1 - y) \log(1 - p)\right], \quad p = \frac{1}{1 + e^{-F}}$$

Iterative Update:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \alpha \cdot h_m(\mathbf{x})$$

where:

- $F_m(\mathbf{x})$ = prediction after m iterations

- α = learning rate
- $h_m(\mathbf{x})$ = m-th tree/weak learner

Part 3: XGBoost (Extreme Gradient Boosting)

Main Ideas of XGBoost

XGBoost (Extreme Gradient Boosting) is an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. It implements machine learning algorithms under the Gradient Boosting framework.

Key Philosophy: XGBoost aims to push the limit of computations resources for boosted tree algorithms. It provides a parallel tree boosting that solves many data science problems in a fast and accurate way.

How XGBoost Differs from Simple Gradient Boosting

Key Differences:

- **Regularization:** Built-in L1 and L2 regularization to prevent overfitting
- **Second-Order Optimization:** Uses both first and second derivatives (Newton's method)
- **System Optimizations:** Column block storage, cache-aware access patterns
- **Sparsity Handling:** Efficient handling of missing values and sparse features
- **Parallelization:** Parallel tree construction and distributed computing
- **Tree Pruning:** Uses max_depth and then prunes trees backward

Mathematical Foundation

XGBoost Objective Function

XGBoost minimizes a regularized objective function:

$$L(\phi) = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k)$$

where:

- $l(y_i, \hat{y}_i)$ = training loss function; Usually, squared error loss for regresssion, and cross-entropy loss for classification.
- $\Omega(f_k) = \gamma T_k + \frac{1}{2}\lambda \sum_{j=1}^{T_k} w_j^2 + \alpha \sum_{j=1}^{T_k} |w_j|$ = regularization term
- T_k = number of leaves in tree f_k
- w_j = leaf weights; It's not a “sample weight” — it's the prediction value that all samples in that leaf will get from that particular tree.

- γ = minimum split loss, λ = L2 regularization parameter, α = L1 regularization parameter

Second-Order Taylor Expansion

XGBoost approximates the loss function using Taylor expansion:

$$L^{(t)} \approx \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

where:

- $g_i = \frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$ = first-order gradient
- $h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial (\hat{y}_i^{(t-1)})^2}$ = second-order gradient (hessian)

Optimal Leaf Weight

The optimal weight for leaf j is:

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

And the corresponding optimal objective reduction is:

$$\text{Obj}^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T$$

How Parallelization Works in XGBoost

Parallel Tree Construction

Key Insight: XGBoost doesn't parallelize tree construction by building multiple trees simultaneously. Instead, it parallelizes the **finding of best split** within a single tree.

Steps in Parallel Split Finding:

1. **Feature-wise Parallelization:** Each thread/process works on different features
2. **Histogram-based Split Finding:** Pre-sort features and use histogram to find splits
3. **Column Block Storage:** Store data in compressed column blocks for cache efficiency
4. **Distributed Computing:** Distribute data across multiple machines

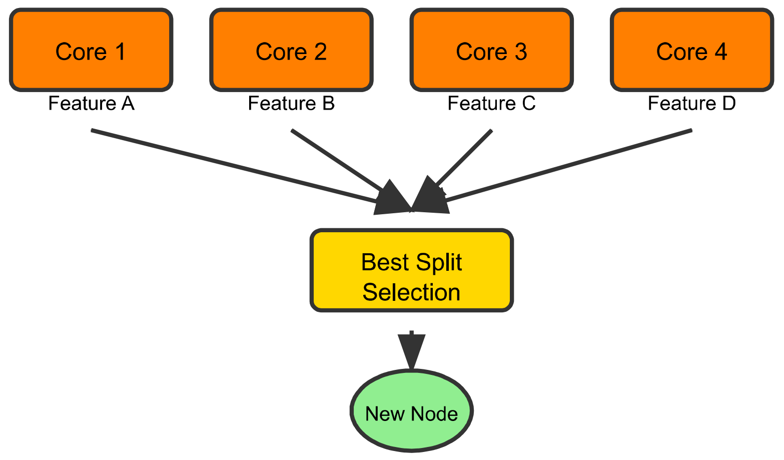
Split Finding Algorithm

For each feature, XGBoost computes the gain for potential splits:

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

This computation can be parallelized across features and candidate split points.

XGBoost Parallel Processing



Part 4: LightGBM (Light Gradient Boosting Machine)

Key Concepts Behind LightGBM

LightGBM is a gradient boosting framework developed by Microsoft that uses tree-based learning algorithms. It's designed to be distributed and efficient with faster training speed and higher efficiency.

Core Philosophy: LightGBM focuses on reducing memory usage and increasing training speed while maintaining high accuracy through novel techniques like leaf-wise tree growth and gradient-based one-side sampling.

How LightGBM Differs from Simple Gradient Boosting

Key Innovations:

- **Leaf-wise Tree Growth:** Grows trees leaf by leaf instead of level by level
- **Gradient-based One-Side Sampling (GOSS):** Keeps large gradient instances, randomly samples small ones
- **Exclusive Feature Bundling (EFB):** Bundles sparse features to reduce feature dimension
- **Histogram-based Algorithm:** Uses histogram-based algorithms for finding splits
- **Memory Efficiency:** Lower memory usage compared to XGBoost
- **Category Features:** Native support for categorical features without encoding

Mathematical Explanation

Gradient-based One-Side Sampling (GOSS)

GOSS keeps instances with large gradients and randomly samples instances with small gradients:

Step 1: Sort instances by absolute value of gradients

Step 2: Keep top $a \times 100\%$ instances with largest gradients

Step 3: Randomly sample $b \times 100\%$ from remaining instances

When computing information gain, amplify small gradient samples by factor $\frac{1-a}{b}$:

$$\text{Gain}_{GOSS} = \frac{[\sum_{i \in A_l} g_i + \frac{1-a}{b} \sum_{i \in B_l} g_i]^2}{n_l^j} - \frac{[\sum_{i \in A_r} g_i + \frac{1-a}{b} \sum_{i \in B_r} g_i]^2}{n_r^j}$$

Exclusive Feature Bundling (EFB)

EFB bundles mutually exclusive sparse features to reduce the number of features:

Graph Coloring Problem: Find minimum bundles such that features in same bundle are mutually exclusive

Conflict Measure: Two features conflict if they are both non-zero for same instance

$$\text{Conflict}(F_i, F_j) = \frac{\#\text{ instances where both } F_i \text{ and } F_j \text{ are non-zero}}{\text{total instances}}$$

Bundle Construction: Use greedy algorithm with conflict threshold γ

Leaf-wise Tree Growth

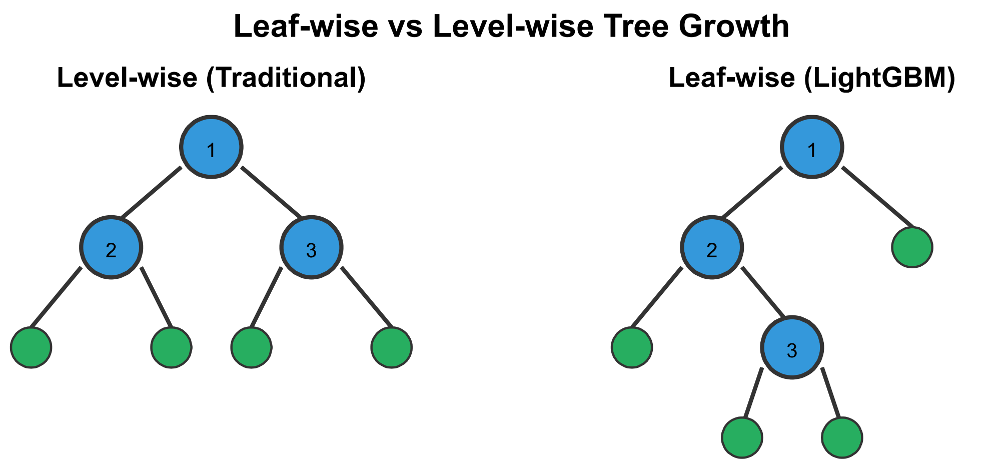
Traditional level-wise growth vs LightGBM's leaf-wise growth:

Level-wise: Split all leaves at same depth

Leaf-wise: Split leaf that reduces loss the most

$$\text{Best Leaf} = \arg\max_{leaf} \Delta\text{Loss}(leaf)$$

This can lead to deeper, more complex trees but better loss reduction per split.



Part 5: CatBoost (Categorical Boosting)

Main Ideas and Key Concepts

CatBoost is a gradient boosting library developed by Yandex that handles categorical features automatically without requiring preprocessing. It's designed to work well out-of-the-box with minimal parameter tuning.

Core Innovation: CatBoost's main breakthrough is its novel approach to handling categorical features using target statistics and ordered boosting to avoid prediction shift and reduce overfitting.

Key Features

CatBoost Innovations:

- **Categorical Features Handling:** Automatic processing without manual encoding
- **Ordered Boosting:** Uses different permutations to compute target statistics
- **Prediction Shift Prevention:** Avoids target leakage in categorical encoding
- **Symmetric Trees:** Uses oblivious decision trees (same split condition at each level)
- **Robust to Overfitting:** Built-in overfitting detection
- **Fast Inference:** Optimized model inference without preprocessing

Mathematical Explanation

Target Statistics for Categorical Features

For categorical feature value $x_{k,i} = c$, CatBoost computes target statistics:

$$\hat{x}_{k,i} = \frac{\sum_{j=1}^{p-1} \mathbb{I}[x_{k,\sigma_j} = c] \cdot y_{\sigma_j} + \alpha}{\sum_{j=1}^{p-1} \mathbb{I}[x_{k,\sigma_j} = c] + \alpha/\text{prior}}$$

where:

- σ = random permutation of training data
- p = position of current example in permutation
- α = smoothing parameter
- prior = prior value (usually average target)

Ordered Boosting

Standard boosting suffers from prediction shift. CatBoost uses ordered boosting:

For each sample x_i:

$$M_i = \text{Model trained on } \{x_j : j < i \text{ in permutation } \sigma\}$$

Residual calculation:

$$r_i = y_i - M_i(x_i)$$

This ensures that the model used to compute residuals for x_i has never seen x_i during training.

Oblivious Decision Trees

CatBoost uses symmetric (oblivious) trees where the same splitting criterion is used across the entire level:

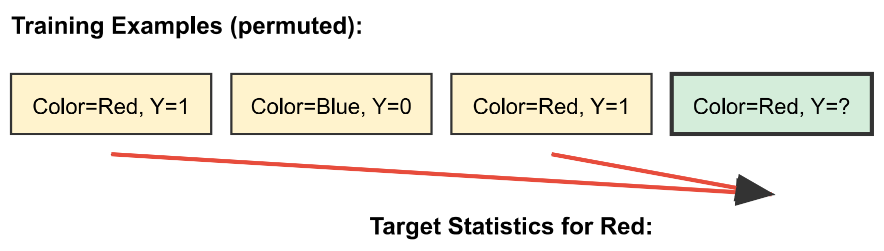
Tree Structure: All nodes at depth d use the same feature and split value

Advantages: Less prone to overfitting, faster scoring, better for GPU

$$\text{Path to leaf} = (f_1(x) > t_1, f_2(x) > t_2, \ldots, f_d(x) > t_d)$$

where f_j and t_j are the same for all nodes at depth j

CatBoost: Ordered Target Statistics



Only uses examples that appear BEFORE current example in permutation

🔑 **Key: No target leakage - current example never used in its own encoding**

Comprehensive Comparison: Gradient Boosting Algorithms

Aspect	Gradient Boosting	XGBoost	LightGBM	CatBoost
Organization	Original algorithm by Jerome Friedman	Developed by Tianqi Chen	Microsoft Research	Yandex
Core Innovation	Sequential boosting with gradient descent	Regularized boosting + system optimizations	Leaf-wise growth + sampling techniques	Categorical feature handling + ordered boosting
Tree Construction	Level-wise growth	Level-wise growth with pruning	Leaf-wise growth	Level-wise with oblivious trees
Optimization Method	First-order gradients	Second-order (Newton) with regularization	First-order with sampling optimizations	First-order with ordered boosting
Regularization	Learning rate only	L1, L2, tree regularization (γ, λ, α)	L1, L2, dropout regularization	Built-in through ordered boosting
Categorical Features	Manual encoding required	Manual encoding required	Some native support	Full automatic handling
Missing Values	Manual handling required	Automatic optimal direction	Automatic handling	Automatic handling
Memory Usage	Moderate	Higher (stores gradients & hessians)	Lower (efficient data structures)	Moderate to low
Training Speed	Baseline	Fast (with parallelization)	Fastest (leaf-wise + GOSS + EFB)	Slower (more computations per tree)

Aspect	Gradient Boosting	XGBoost	LightGBM	CatBoost
Parallel Processing	Limited	Feature-wise parallelization	Feature and data parallelization	Multi-threading support
GPU Support	No	Yes (CUDA)	Yes (OpenCL, CUDA)	Yes (CUDA)
Overfitting Risk	Moderate	Lower (regularization)	Higher (leaf-wise growth)	Lowest (ordered boosting)
Hyperparameter Tuning	Moderate effort	Requires careful tuning	Requires careful tuning	Minimal tuning needed
Key Algorithms	MART, GBDT	Exact greedy, approximate histogram	GOSS, EFB, histogram-based	Ordered TS, oblivious trees
Mathematical Foundation	$\min L(y, F)$	$\min L(y, F) + \Omega(f)$	$\min L(y, F)$ + sampling	$\min L(y, F)$ + ordered statistics
Best Use Cases	Educational, simple problems	Tabular competitions, structured data	Large datasets, speed critical	High cardinality categoricals, robust models
Typical Learning Rate	0.1	0.1 - 0.3	0.1	0.03 - 0.1
Max Tree Depth	6-10	3-10	Deep trees (15-31)	6-10 (oblivious)
Popular Libraries	scikit-learn	xgboost	lightgbm	catboost

Aspect	Gradient Boosting	XGBoost	LightGBM	CatBoost
Interpretability	Good	Good (SHAP integration)	Good (built-in plots)	Excellent (feature importance)
Production Deployment	Simple	Good ecosystem	Lightweight models	Self-contained, no preprocessing

🎯 Selection Guidelines:

- **Choose Gradient Boosting:** Learning, understanding fundamentals, simple datasets
- **Choose XGBoost:** Competitions, well-preprocessed data, need maximum accuracy
- **Choose LightGBM:** Large datasets, speed is critical, memory constraints
- **Choose CatBoost:** Many categorical features, robust out-of-the-box performance, minimal preprocessing