📌 Extreme Gradient Boosting (XGBoost)

✅ What is XGBoost?

- **XGBoost is not an algorithm**, it's a **library** built on top of the Gradient Boosting framework.
- It combines:
  - Machine Learning: **Gradient Boosting**
  - Engineering: **Performance Optimization, Parallelism, Hardware Utilization**

🎯 Why XGBoost Uses Gradient Boosting as Base

1. Flexibility

- Can use **any differentiable loss function**.
- Works for:
  - Regression
  - Classification (binary/multiclass)
  - Time series
  - Ranking
  - Anomaly detection

2. Performance

- Fast due to engineering innovations.

3. Robustness

- Can handle **missing values internally**.

🚀 Features of XGBoost

1. ⚙️ Flexibility

- **Cross-Platform:** Windows, Linux, macOS
- **Languages:** Python, R, Julia, Java, Scala, etc.
- **Integrations:** scikit-learn, Dask, Spark, MLFlow, Kubernetes, SHAP, LIME, Airflow

---

2. ⚡ Speed Optimizations

(a) Parallel Processing (During Tree Building)

- Trees are built sequentially.

- But **feature splitting is done in parallel** across multiple cores.

(b) Optimized Data Structure

- Uses **Column Block format** (stores data column-wise).
- Improves **cache efficiency**.

(c) Cache Awareness

- **Histogram-based training** stores frequently accessed bin values in cache.
- Improves speed by reducing memory access time.

(d) Out-of-Core Computing

- Useful for very large datasets.
- Loads and trains on **data in chunks**.
- Memory-efficient, but works **on a single machine** sequentially.

(e) Distributed Computing

- Data and tasks are split across **multiple machines**.
- All machines compute their part → a **master node aggregates results**.
- Faster training and handling of massive datasets.

(f) GPU Support

- Tree method: `gpu_hist`
- Leverages many weak but parallel GPU cores for faster computation.

---

3. 🔧 Performance Features

- **Regularized Learning Objective:** Prevents overfitting (L1 & L2 regularization)
- **Handles Missing Values:** Automatically splits on available data
- **Sparsity-aware Splits:** Deals with sparse input data
- **Approximate Split Finding (Sketching):** For large datasets
- **Tree Pruning:** Removes branches that do not improve performance

---

🧠 Summary

Feature	Description
Base	Gradient Boosting Decision Tree (GBDT)
Loss Functions	Any differentiable loss (custom or standard)
Speed	Fast due to parallel feature splitting, histogram binning, GPU support
Scalability	Out-of-core (1 machine) & Distributed (multi-machine) supported
Integration	With many platforms, cloud tools, and languages

Feature	Description
Handling Missing Data	Automatically handled without preprocessing
Interpretability	Tools like SHAP, LIME supported

🖇️ Common Tree Methods in XGBoost

Tree Method	Description
`exact`	Exact greedy algorithm
`approx`	Approximate algorithm (sketching)
`hist`	Fast histogram optimized algorithm
`gpu_hist`	GPU accelerated histogram algorithm

📈 XGBoost for Regression

🧠 Overview

XGBoost for regression follows the **same boosting principle** as Gradient Boosting:

1. Start with a **base model** (mean of the target).
2. Train a sequence of **Decision Trees** to learn the **residuals** (errors).
3. Update the model by adding the new tree's predictions.

However, the **tree-building logic in XGBoost** differs significantly:

- It does **not use Gini or Entropy** (like classification trees).
- It uses a **custom objective function** involving **residuals** and **similarity scores**.

🪄 Tree Formation in XGBoost

Step 1: Leaf Node Formation

Before building the tree, for each **leaf node**, we calculate a **Similarity Score (SS)**:

$$SS = \frac{(\sum \text{residuals})^2}{n + \lambda}$$

Where:

- (n): number of residuals in the leaf
- (\lambda): regularization parameter (prevents overfitting)

Step 2: Gain Calculation

When trying to split a node, we calculate the **gain** (improvement in similarity):

$$\text{Gain} = SS_{\text{Left}} + SS_{\text{Right}} - SS_{\text{Parent}} - \gamma$$

Where:

- (\gamma): regularization term to penalize excessive tree depth (also helps pruning)

A split is accepted **only if the gain is positive**.

Step 3: Tree Building Logic

- XGBoost splits based on **maximizing Gain**, not reducing entropy or Gini.
- Trees are grown greedily using:
  - **Exact Greedy Algorithm** → for **small datasets**
  - **Approximate Algorithm** → for **large datasets**

🌿 Output at Each Leaf Node

Once the best splits are found and leaf nodes are created, the output value for each leaf is computed as:

$$\text{Output}_{\text{leaf}} = \frac{\sum \text{residuals}}{n + \lambda}$$

This output becomes the **prediction from that leaf node**.

🔁 Updating the Model (Stage 2)

After the first tree (model), we update the prediction:

$$F_1(x) = \text{mean} + \eta \cdot f_1(x)$$

- (\eta): learning rate (default in XGBoost is 0.3)
- (f_1(x)): prediction from first decision tree

Then, compute **new residuals**:

$$\text{Residual}_2 = y - F_1(x)$$

Train the next decision tree on these residuals.

🔄 Summary of Tree Formation Steps

1. Start with **mean prediction**.

2. Compute **residuals**.
3. Use residuals to compute **similarity score** for potential splits.
4. Use **gain** to decide best split.
5. Continue growing tree using **greedy or approximate algorithms**.
6. Update model using:

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

🧪 Tree Building Algorithms

Method	Description	Use Case
Exact Greedy	Tries all splits	Small datasets
Approximate	Uses histogram/binning technique	Large datasets

🚀 XGBoost for Classification

🔍 Core Idea

In classification using XGBoost, we start with a **base model** that estimates the **log-odds**, and then build decision trees sequentially to minimize the error.

Base Model (Model 1):

$$\text{Log(odds)} = \log_e\left(\frac{P}{1-P}\right)$$

- (P): Probability of the positive class.
- This log-odds is the starting prediction (similar to how mean is used in regression).

After calculating log-odds, we convert them into **probabilities**:

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

🌳 Decision Tree Formation (Model 2 onward)

All subsequent models are decision trees trained on **residuals**, similar to gradient boosting.

📌 Similarity Score for Classification

XGBoost uses a different formula for classification vs regression. For a given leaf node, the **similarity score (SS)** is:

$$SS = \frac{(\sum \text{residuals})^2}{\sum (p_i(1-p_i)) + \lambda}$$

Where:

- (p_i): predicted probability for the (i)-th data point
- (\lambda): regularization parameter
- Residual = actual - predicted (for classification)

🎯 Goal of Tree Building

- Try different splits in input columns.
- For each potential split, compute **gain** based on change in similarity score:

$$\text{Gain} = SS_{\text{Left}} + SS_{\text{Right}} - SS_{\text{Parent}} - \gamma$$

- Choose the split that maximizes this gain.
- Repeat to grow tree until stopping criteria is met.

🌿 Leaf Output Calculation

Once a node is created, calculate its output (log-odds shift):

$$\text{Output}_{\text{leaf}} = \frac{\sum \text{residuals}}{\sum p_i(1-p_i) + \lambda}$$

🔁 Update Step

1. Add leaf output to the **log-odds**.
2. Convert updated log-odds to **probability**:

$$P = \frac{e^{\text{new log(odds)}}}{1 + e^{\text{new log(odds)}}}$$

3. Compute new residuals.
4. Train next tree on updated residuals.

✅ Assumptions

1. **Weak Learners Can Be Combined**:
   Multiple weak models (e.g., shallow trees) can be improved sequentially.

2. **Additive Modeling Framework**:
   Improvements from each tree combine additively to improve the model.

3. **Target Variable is Predictable**:
   Assumes that meaningful patterns exist in input features.

4. **Proper Feature Engineering**:
   Features must be informative and clean (low noise).

5. **Low Multicollinearity**:
   High correlation among features may degrade performance.

⚠️ Limitations

1. **High Computational Demand**
   - Training is slow and memory-intensive for large/high-dimensional data.
2. **Risk of Overfitting**
   - Too many deep trees can overfit without proper tuning or regularization.
3. **Hyperparameter Complexity**
   - Numerous parameters (learning rate, depth, gamma, etc.) require expert tuning.
4. **Less Interpretability**
   - Complex ensemble of trees is harder to explain than single decision trees or logistic regression.
5. **Sensitive to Noise**
   - Noisy datasets can mislead the model unless early stopping or regularization is applied.
6. **Handling Imbalanced Datasets**
   - Requires special strategies like custom loss functions or class weighting.
7. **Scaling Challenges with Sparse Data**
   - Performs better than traditional models on sparse data, but preprocessing is still crucial.
8. **Training Complexity**
   - Boosting is sequential in nature, which limits full parallelization (though XGBoost uses optimized techniques like histogram-based training).

🔧 XGBoost Hyperparameter Guide

XGBoost is a powerful and flexible boosting algorithm that provides a wide range of parameters to tune performance for **both classification and regression** tasks. Here's a breakdown of the most important hyperparameters:

1️⃣ General Parameters

Parameter	Description
`booster`	Type of model to run: • `'gbtree'`: tree-based models (default) • `'gblinear'`: linear models • `'dart'`: Dropout-based boosting
`nthread` or `n_jobs`	Number of CPU threads to use for training. Useful for parallel processing.

2️⃣ Booster Parameters (for `'gbtree'` and `'dart'`)

These parameters control the trees and boosting behavior.

🌳 Tree Structure

Parameter	Description
`max_depth`	Maximum depth of a tree. Higher = more complex trees. Prevents underfitting.
`min_child_weight`	Minimum sum of instance weights needed in a child. Higher values prevent overfitting.
`gamma`	Minimum loss reduction required to make a split. Larger gamma = more conservative trees.
`subsample`	% of training samples used for growing each tree. Typical values: 0.5 to 1.0
`colsample_bytree`	% of features (columns) used per tree. Can reduce overfitting.
`colsample_bylevel`	% of features used per tree level.
`colsample_bynode`	% of features used per split (node).
`lambda` (reg_lambda)	L2 regularization on weights. Helps with overfitting.
`alpha` (reg_alpha)	L1 regularization on weights. Adds sparsity.

3️⃣ Learning Parameters

Parameter	Description
`learning_rate` (or `eta`)	Step size shrinkage used in updates. Lower values make learning slower but more robust. Typical range: 0.01–0.3
`n_estimators`	Number of trees (boosting rounds).

Parameter	Description
`scale_pos_weight`	Used for imbalanced classification. Ratio of negative to positive classes.

4️⃣ Objective Parameters

Parameter	Description
`objective`	Learning task type: • `'binary:logistic'` : binary classification • `'multi:softmax'` : multiclass classification (with `num_class`) • `'multi:softprob'` : same as softmax, but returns probabilities • `'reg:squarederror'` : regression (default) • `'reg:logistic'` : logistic regression
`eval_metric`	Evaluation metric: • Classification: `'logloss'` , `'error'` , `'auc'` • Regression: `'rmse'` , `'mae'` , `'rmsle'`

5️⃣ Dart-Specific Parameters (Optional)

Parameter	Description
`sample_type`	Sampling method: `'uniform'` or `'weighted'`
`normalize_type`	How to normalize dropped trees: `'tree'` or `'forest'`
`rate_drop`	Dropout rate for trees
`skip_drop`	Probability of skipping the dropout during iteration

🧠 Typical Settings

🔵 Binary Classification

```
params = {
    'objective': 'binary:logistic',
    'eval_metric': 'logloss',
    'max_depth': 4,
    'learning_rate': 0.1,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'scale_pos_weight': 1,
    'n_estimators': 100
}
```

```
params = {
    'objective': 'reg:squarederror',
    'eval_metric': 'rmse',
```

```
    'max_depth': 4,
    'learning_rate': 0.1,
    'subsample': 0.9,
    'colsample_bytree': 0.9,
    'n_estimators': 100
}
```

In [ ]: