**Boosting Theory**

## **What is Boosting?**

Boosting is an ensemble learning technique that combines many **weak learners** (usually decision trees) to create a strong learner.

* A weak learner is a simple model that performs slightly better than random guessing (e.g., a shallow decision tree).
* Boosting builds models **sequentially**, where each new model tries to fix the mistakes of the previous ones.

## **Key Idea**

1. Start with a simple model (weak learner).
2. Identify the samples that were misclassified.
3. Give more weight to those hard-to-classify samples.
4. Train the next model focusing on these harder samples.
5. Combine all weak learners into one strong model.

**Difference vs Bagging**

* **Bagging** (Bootstrap Aggregating): trains models in **parallel**, averages results → reduces variance.
* **Boosting**: trains models **sequentially**, each improves on previous → reduces bias.

## **Popular Boosting Algorithms**

1. **AdaBoost (Adaptive Boosting)**
   1. Adjusts weights of samples: misclassified → higher weight.
   2. Final prediction = weighted vote of all learners.
2. **Gradient Boosting**
   1. Builds new models to predict the **residual errors** (difference between actual & predicted).
   2. Uses gradient descent to minimize loss.
3. **XGBoost (Extreme Gradient Boosting)**
   1. Optimized version of Gradient Boosting (faster, handles missing data, regularization).
4. **LightGBM**
   1. Gradient boosting using **leaf-wise tree growth** (faster, memory efficient).
5. **CatBoost**
   1. Gradient boosting optimized for **categorical features**.

**AdaBoost**

## **Core Idea**

* Build **multiple weak classifiers** (usually Decision Stumps = trees with depth=1).
* Each classifier focuses on the **samples misclassified** by the previous ones.
* Misclassified points get **higher weights**, so the next learner pays more attention to them.
* Final prediction = **weighted majority vote** (classification) or weighted sum (regression).

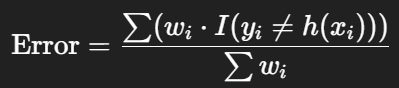
## **How AdaBoost Works (Algorithm)**

* **Initialize weights**

Each training sample gets equal weight

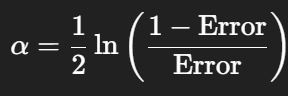


* **Train a weak learner** (e.g., decision stump).
* **Calculate error**



where I() is an indicator function.

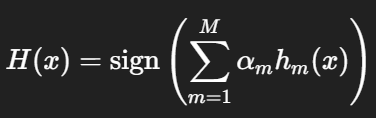
* **Compute model weight** (importance of learner)

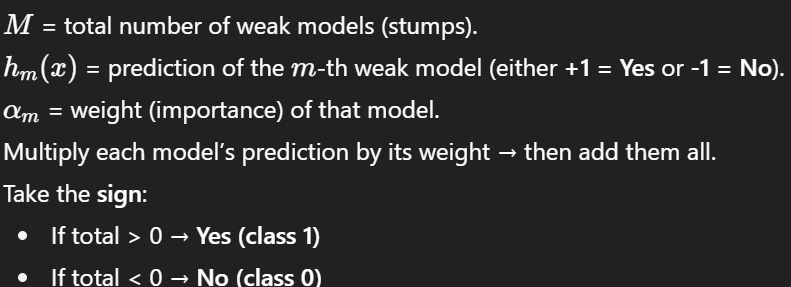


* **Update sample weights**
  + Misclassified samples → weight increases.
  + Correctly classified samples → weight decreases.



* **Normalize weights** so they sum to 1.
* **Repeat** steps 2–6 for *M* iterations.
* **Final Prediction**





## **Intuition**

* First classifier: weak, makes many mistakes.
* Second classifier: focuses on correcting those mistakes.
* Over time: ensemble becomes **much stronger**.

## **Pros**

* Works well for binary classification.
* Reduces bias significantly.
* Simple to implement.

## **Cons**

* Sensitive to noise and outliers.
* Can overfit if too many learners.

**Gradient Boosting**

## **What is Gradient Boosting?**

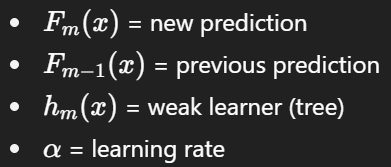
* Gradient Boosting is an ensemble method that builds models sequentially.
* Each new model tries to correct the errors made by the previous models.
* Uses decision trees (usually shallow ones) as weak learners.
* Instead of simple voting/averaging (like Bagging/Random Forests), it optimizes a loss function using gradient descent.

## **How it Works (Step-by-Step)**

1. Start with a base prediction (often the mean of target values for regression, or log odds for classification).
2. Compute the errors (residuals) between the prediction and actual values.
3. Fit a small decision tree to predict these residuals.
4. Update the prediction:



where:



1. Repeat until the error stops improving.

## **Key Concepts**

* **Weak Learner:** Usually, a decision tree with max depth 1 (stump) or 2.
* **Learning Rate (α):** Shrinks the contribution of each new tree (prevents overfitting).
* **Number of Trees (**n\_estimators**):** More trees → better fit, but risk of overfitting.
* **Loss Function:** Guides the optimization.
  + Regression: Mean Squared Error (MSE).
  + Classification: Log Loss / Deviance.

## **Advantages**

* Handles both regression & classification.
* Works well with structured/tabular data.
* Can handle different loss functions.
* Robust to outliers (if tuned properly).

## **Disadvantages**

* Slower training compared to Random Forest.
* Sensitive to hyperparameters (learning rate, depth, trees).
* Not great for extremely high-dimensional sparse data (like raw text).

## **Important Hyperparameters**

* n\_estimators: Number of boosting rounds (trees).
* learning\_rate: Step size shrinkage (default ~0.1).
* max\_depth: Depth of trees.
* subsample: Fraction of samples used per tree (helps reduce overfitting).
* min\_samples\_split / min\_samples\_leaf: Minimum samples to split/leaf.

**XGBoost**

## **What is XGBoost?**

* XGBoost (Extreme Gradient Boosting) is an advanced implementation of gradient boosting, designed for speed, efficiency, and accuracy.
* At its core, it’s still Gradient Boosting — building trees sequentially, each trying to fix the errors of the previous one.  
   But XGBoost makes this process faster and more regularized.

**XGBoost vs Gradient Boosting:**

* XGBoost has built in regularization (reg\_alpha, reg\_lambda)
* Can handle missing values.
* Is faster than gradient boosting
* Built in cross validation capability.
* Tree grows depth wise so reduces loss faster.
* Performs parallelization

**How does XGBoost performs parallelization:**

* Normal Gradient Boosting: checks split one by one (serial).
* XGBoost: checks splits for many features at the same time (parallel) using multiple CPU cores.
  + Core 1 checks Age feature.
  + Core 2 checks Income.
  + Core 3 checks Education.
  + … then it picks the best split overall.
* So → trees are still built one after another, but within each tree, split finding is parallelized, making training much faster.

**How regularization happens in XGBoost:**

* Regularization in XGBoost (through reg\_alpha and reg\_lambda) encourages the leaf predictions to be smaller right from the training step, so the model doesn’t rely on extreme values.
* Adds a penalty to the objective function during training.
* This penalty discourages leaves from having very large values in the first place.
* So instead of shrinking *after prediction*, it actually changes how the leaf values are chosen during training.

**Hyperparameter Tuning for Ensembles**

Ensemble methods (Bagging, Boosting, Stacking) combine multiple weak learners to create a stronger model.  
 But their performance **depends heavily on hyperparameters** → tuning is crucial.

1. **Bagging (Bootstrap Aggregating) / Random Forest**

Bagging builds many models in parallel on random subsets.

**Key hyperparameters to tune:**

* n\_estimators **:** number of trees.
  + Too few → underfit. Too many → slow but stable.
* max\_features: number of features each split sees.
  + Small = more diversity, less correlation.
* max\_samples: number of data samples per tree.
* max\_depth, min\_samples\_split, min\_samples\_leaf: control tree complexity.
* bootstrap: whether to sample with replacement (default True).

**Typical tuning:**

param\_grid = {

"n\_estimators": [100, 300, 500],

"max\_depth": [5, 10, None],

"max\_features": ["sqrt", "log2"],

"min\_samples\_split": [2, 5, 10]

}

**2. Boosting (AdaBoost, GradientBoosting, XGBoost, LightGBM, CatBoost)**

Boosting builds models sequentially, each correcting errors of the previous.

**General hyperparameters:**

* n\_estimators: number of boosting rounds (trees).
* learning\_rate: shrinkage factor (small = stable but needs more trees).
* max\_depth: depth of trees (higher = more complex).
* subsample: fraction of data per tree (helps regularize).
* colsample\_bytree: fraction of features per tree (for XGBoost/LightGBM).
* reg\_alpha, reg\_lambda: L1/L2 regularization (XGBoost).
* min\_child\_weight (XGBoost/LightGBM): min sum of weights in a leaf.

**Example tuning (XGBoost):**

param\_grid = {

"n\_estimators": [100, 300],

"learning\_rate": [0.01, 0.1, 0.3],

"max\_depth": [3, 5, 7],

"subsample": [0.8, 1.0],

"colsample\_bytree": [0.8, 1.0],

"reg\_alpha": [0, 0.1, 1],

"reg\_lambda": [1, 5, 10]

}

### **GridSearchCV**

1. You define exact values for each hyperparameter.
2. It tries all possible combinations (exhaustive search).

**Example:**

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier

param\_grid = {

"n\_estimators": [100, 200, 300], # fixed values

"max\_depth": [5, 10, 15] # fixed values

}

grid = GridSearchCV(

RandomForestClassifier(),

param\_grid,

cv=3, scoring="accuracy"

)

grid.fit(X\_train, y\_train)

print("Best Params:", grid.best\_params\_)

Total = 3 × 2 = 6 models trained.

Pros: systematic, guarantees best in grid.  
Cons: very slow if grid is large.

### **RandomizedSearchCV**

* You **define ranges / distributions** (not fixed values).
* It tries a **random subset** of combinations.

**Example:**

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.ensemble import RandomForestClassifier

from scipy.stats import randint

param\_dist = {

"n\_estimators": randint(50, 500), # random int between 50–500

"max\_depth": randint(3, 20) # random int between 3–20

}

rand = RandomizedSearchCV(

RandomForestClassifier(),

param\_distributions=param\_dist,

n\_iter=10, # only try 10 random combos

cv=3, scoring="accuracy", random\_state=42

)

rand.fit(X\_train, y\_train)

print("Best Params:", rand.best\_params\_)