XGBoost

XGBoost (Extreme Gradient Boosting) is a highly efficient implementation of the Gradient Boosting framework, widely used for supervised machine learning tasks such as regression and classification. It also works well for ranking problems and time-series problems. XGBoost stands out because it introduces several optimizations over traditional Gradient Boosting that significantly improve both training speed and model performance.

Why XGBoost was built on top of Gradient Boost?

XGBoost was built on top of Gradient Boosting because of its flexibility, proven effectiveness, and natural compatibility with decision trees. Gradient Boosting's sequential learning process, ability to optimize any differentiable loss function, and strong emperical performance across a wide range of tasks made it an ideal foundation for XGBoost.

Key Features of XGBoost:

The features of XGBoost can be widely divided into three major categories —

I. Flexibility:

(a) **Cross Platform:** The XGBoost library is available to run on any platform like Windows, Ubuntu, Mac, Linux, etc.

(b) **Multiple Language Support:** XGBoost has built-in wrappers which allows programmers to use XGBoost in multiple programming languages like Python, R, Scala, Java, Ruby, Julia, Matlab, etc.

(c) **Integration with other libraries and tools:** XGBoost is compatible for integrating with other libraries and frameworks and tools like Numpy, Scikit-learn, pandas, Pyspark, Dask, SHAP, Lime, etc.

(d) **Supports all kinds of ML problems:** It can be used to solve problems like regression, classification, time-series forecasting, ranking problems, custom problems as well.

II. **Speed:** XGBoost is highly optimized for speed, making it a preferred choice for large-scale machine learning tasks. Several features contribute to its speed:

(a) **Parallel processing:** In boosting the trees are added sequentially to reduce the errors made by the previous model. The parallelization in XGBoost comes from the fact that each the construction of each decision tree can be parallelized. XGBoost achieves parallelism primarily during the process of constructing individual decision trees. Specifically, the process of finding the best splits for each tree can be done in parallel. At each node, the algorithm needs to find the best feature and value to split the data. XGBoost can evaluate the potential splits for different features in parallel.

(b) **Optimized data structures:** XGBoost stores data in a compressed columnar format called the "column block". This format allows efficient access to the feature columns and enables parallel processing of features when building the tree. The use of column blocks reduces memory overhead and speeds up the computation of split criterion.

(c) **Cache awareness:** XGBoost uses histogram-based binning as an approximation method to speed up the process of finding the best split in decision tree construction. In histogram-based binning, continuous feature values are grouped into discrete intervals (bins). Instead of evaluating all possible split points for continuous features, XGBoost evaluates splits based on these pre-defined bins. XGBoost maintains these histograms in memory (cached) to quickly access and reuse them when needed. By caching the histograms, XGBoost significantly reduces the number of operations required for split evaluation, leading to faster tree construction.

(d) **Out of core computing:** XGBoost supports out-of-core computing, which allows it to handle datasets that is too large to fit into memory. By processing data in chunks

and using disk space and caching efficiently, XGBoost can scale to very large datasets while maintaining speed.

(e) **Distributed Computing:** XGBoost supports distributed computing, allowing it to scale across multiple machines. This is crucial for big data scenarios where a single machine might struggle with memory limitations or processing time.

(f) **GPU Support:** XGBoost has support for using GPUs for making the computations faster. We can use tree method as- 'gpu-hist' in the XGBoost library to use this feature.

II. **Performance:** XGBoost is known for achieving state-of-the-art performance in various machine learning competitions and real world application due to several key enhancement made in Gradient Boosting algorithm. Some of these are—

(a) **Regularized learning objective:** XGBoost introduces regularization terms in the objective function, which helps to penalize model complexity and prevent overfitting. L1 regularization encourages sparsity in the model and L2 regularization penalizes large coefficients. Due to this regularization capability XGBoost outperforms traditional gradient boosting methods, especially in noisy data or dataset with high dimensional features.

(b) **Handling Missing Values:** XGBoost natively handles missing data by learning the best direction to take at a node when a feature is missing. This eleminates the need for explicit imputation methods.

(c) **Sparsity aware split finding:** XGBoost has built in support for sparse data. It can automatically handle missing values and sparse data by learning which direction (left or right) to take in the decision tree.

(d) **Efficient Split finding (Weighted quantile sketch + Approximate Tree Learning):** For efficient handling of continuous variables, XGBoost uses a weighted quantile

sketch algorithm, which allows it to handle weighted data points and more accurate splits. Also, XGBoost uses an approximate greedy algorithm to split nodes when dealing with large datasets. Instead of evaluating every possible split (as in traditional gradient boosting), it uses the quantile sketch algorithm to quickly estimate candidate splits. This reduces the time complexity of tree construction, making XGBoost much faster for larger datasets.

(e) **Tree Pruning:** XGBoost uses a technique called 'Max Depth Pruning'. This technique stops the construction of the tree when it reaches a certain maximum depth or when splitting a node no longer results in a significant improvement in the model's performance. We have a parameter named 'gamma' which can be used in XGBoost.

(f) **Second order Taylor approximation:** XGBoost improves optimization by using both first and second order derivatives (gradients and Hessians) of the loss function. The second order approximation allows for more precise updates to the model's weights, leading to faster convergence compared to traditional Gradient Boosting, which uses only first-order gradients.

XGBoost for Regression :

XGBoost performs regression tasks in the same manner as it was performed in Gradient Boost, where the first model that is built predicts all the outcomes as the mean and then subsequent models that are build are decision trees which build trees on residuals of the previous models. The trees built in XGBoost are different than normal decision trees because here the splits in trees are not decided using gini impurity / entropy. Instead, here in XGBoost we use similarity scores and we try to divide the nodes such that it leads in increasing the similarity score. Let us understand this using an example —

Input Output Mean Residual of model 1

Eg:1

cgpa	package	model(M_1)	Res 1
6.7	4.5	7.3	-2.8
9.0	11.0	7.3	3.7
7.5	6.0	7.3	-1.3
5.0	8.0	7.3	0.7

Here, we have two columns cgpa and package, where we want to predict package using the cgpa column.

From gradient boosting, we know the first model is a simple mean of all the elements in output column and then we calculate the residual by taking the difference of output column and residual mean column. In step2, we will make a regression tree using the input (cgpa) column and residual (res1) column. We start by creating a leaf node which has all the residuals and we calculate an initial similarity score. The formula for calculating similarity score is given as—

Step 2:

** Similarity score $= \frac{(\text{Sum of all residuals})^2}{\#\text{Of residuals} + \lambda} = \frac{\left(\sum_{i=1}^{n} \text{res}_i\right)^2}{n + \lambda}$

leaf node is ↓

Here, $\boxed{-2.8, 3.7, -1.3, 0.7}$ λ → hyperparameter

to calculate similarity score $= \frac{(-2.8+3.7-1.3+0.7)^2}{4+0}$ → taking $\lambda = 0$

$SS_{root} = 0.02$

Step 3: Now, we go for splitting the nodes in the same manner we did for gradient boosting trees by arranging the column in ascending order and then getting intermediate points for splitting. For each split point we split the dataset and calculate the gain in importance/similarity score. The split which has maximum similarity score gets choosen as the point for making initial split.

cgpa	res1
5.0	0.7
6.7	-2.8
7.5	-1.3
9.0	3.7

5.85, 7.1, 8.25

Split 1: cgpa <= 5.85

Yes: 0.7 → $SS_{left} = \frac{(0.7)^2}{1}$

No: -2.8, -1.3, 3.7 → $SSR = \frac{(-2.8-1.3+3.7)^2}{3}$

$SSR = \frac{0.16}{3}$

$SS_{left} = 0.49$

$SS_{Right} = 0.05$

** $\text{Gain} = SS_L + SS_R - SS_{Root}$

$= 0.49 + 0.05 - 0.02$

$\text{Gain} = 0.52$

Split 2: cgpa <= 7.1

Yes: 0.7, -2.8 → $SS_L = \frac{(0.7-2.8)^2}{2}$

$SS_L = \frac{4.41}{2}$

$SS_L = 2.20$

No: -1.3, 3.7 → $SS_R = \frac{(-1.3+3.7)^2}{2}$

$SS_R = \frac{5.76}{2}$

$SS_R = 2.88$

$\text{Gain} = 2.20 + 2.88 - 0.02$

$\text{Gain} = 5.06$

Split 3: cgpa <= 8.25

Yes: 0.7, -2.8, -1.3 → $SS_L = \frac{(0.7-2.8-1.3)^2}{3}$

$SS_L = \frac{11.56}{3}$

$SS_L = 3.85$

No: 3.7 → $SS_R = (3.7)^2$

$SS_R = 13.69$

$\text{Gain} = 3.85 + 13.69 - 0.02$

$\text{Gain} = 17.52$

Since gain in similarity was highest in split 3, we will use that for making our first node. Now, we need to split the left sub-tree which has three points, we will again repeat the process as—

$SS_{Root} = (0.7-2.8-1.3)^2/3 = 3.085$

cgpa	Res1
5.0	0.7
6.7	-2.8
7.5	-1.3

Split 1: cgpa <= 5.85

Yes: 0.7 — $SS_L = (0.7)^2/1$, $SS_L = 0.49$

No: -2.8, -1.3 — $SS_R = (-2.8-1.3)^2/2$, $SS_R = 8.4$

Gain = 0.49 + 8.4 − 3.085

Gain = 5.04 ✓

cgpa <= 7.1

Yes: 0.7, -2.8 — $SS_L = (0.7-2.8)^2/2$, $SS_L = 2.20$

No: -1.3 — $SS_R = (-1.3)^2/1$, $SS_R = 1.69$

Gain = 2.20 + 1.69 − 3.085

Gain = 0.04 ✗

We will keep height/max-depth for our tree as 2, so our tree looks like this for now.

Stage 2 Model ⇒ Stage 1 + lr × model 0.3

cgpa <= 8.25, SS = 0.02, Gain = (17.52)

Yes: cgpa <= 5.85, SS = 3.085, Gain = 5.04

- Yes: 0.7 — Output = 0.7/1 = 0.7
- No: -2.8, -1.3 — Output = (-2.8-1.3)/2 = -2.05

No: 3.7, SS = 13.69 — Output = 3.7/1 = 3.7

Now, for these leaves, the output is calculated using formula—

$$\text{Output} = \frac{\text{Sum of residuals}}{\#\text{residuals} + \lambda} \quad **$$

In XG Boost, default learning rate = 0.3

cgpa	Salary	(M_1)	res 1	M_2	res 2	learning rate
6.7	4.5	7.3	-2.8	6.69	-2.19	⇒ 7.3 + 0.3 * (-2.05) = 6.69
9.0	11.0	7.3	3.7	8.41	2.59	⇒ 7.3 + 0.3 * (3.7) = 8.41
7.5	6.0	7.3	-1.3	6.69	-0.69	⇒ 7.3 + 0.3 * (-2.05) = 6.69
5.0	8.0	7.3	0.7	7.51	0.49	⇒ 7.3 + 0.3 * (0.7) = 7.51

From the res2, we can see as we add more models, the residuals decrease, thus indicating we are moving in the same correct direction as our goal is to reduce the residual error.

We can add even more trees, now by using cgpa column and res2 column in the similar manner we did above.

* One thing to note is that when we have similar elements in the leaf our similarity score will be high, if two elements are of opposite magnitude then they cancel each other reducing similarity score.

Pruning of the branches in regression trees

We take a value threshold γ (gamma), let's say it was 10, then we start pruning the tree from the bottom, we calculate the difference of γ with gain and if it comes as negative we can prune that branch. For Eg: Gain from split criteria (cgpa <= 5.85) was 5.04 and our $\gamma = 10$, hence we can get negative in difference, thus that branch can be pruned. After cutting that split, we go one level up and check Gain for split criteria (cgpa <= 8.25) was 17.52, here difference will be positive then it will not be pruned. Suppose for the bottom branch difference is positive and for upper branch difference between γ and importance gain is negative, then also we can't prune as bottom is +ve.

XGBoost for Classification.

For solving classification problems, XGBoost more or less works the same way as done in Gradient Boosting but with a few minor changes. Lets have a look at a simple example to see how it works, we are trying to predict whether a student will get placed given a data of his cgpa.

Step 1:

cgpa	placed	M_1 = log odds	res 1
5.7	0	0.405	-0.6
6.25	1	0.405	0.4
7.1	0	0.405	-0.6
8.15	1	0.405	0.4
9.60	1	0.405	0.4

Similar to Gradient Boost the initial model (M_1) is nothing but simple log odds of the student getting placed for each output. Here, log odds (placed) = $\log_e\left(\frac{3}{2}\right) = 0.405$

To calculate the residuals we need to have probability, and probability is calculated using the formula -

* $p = \frac{1}{1+e^{-\text{log odds}}} = \frac{1}{1+e^{-0.405}} = 0.6$, now we can calculate the residual of our initial prediction by subtracting target column with probabilities.

Step 2: Now, we will add a decision tree model to our initial model. We will use a regression tree by using input column (cgpa) and output as the residual column (res 1). We will create a tree of max depth = 2, so as to avoid overfitting in this small dataset.

In classification problems:

$$SS = \frac{\Sigma(\text{residuals}_i)^2}{\text{prevprob}(1-\text{prevprob}) + \lambda}$$ → for now $\lambda = 0$

$$SS_{Root} = \frac{(-0.6+0.4-0.6+0.4+0.4)^2}{5*[0.6(1-0.6)]+0}$$

$SS_{Root} = 0$

cgpa	res1
5.70	-0.6
6.25	0.4
7.10	-0.6
8.15	0.4
9.60	0.4

Since, our cgpa column is already sorted in ascending order, we will create intermediate split criterions using which we can split

Split 1: cgpa <= 5.97

Yes → -0.6 ; No → 0.4, -0.6, 0.4, 0.4

$SS_L = \frac{(-0.6)^2}{1*0.6*0.4}$ $SS_R = \frac{(0.4-0.6+0.4+0.4)^2}{4*0.24}$

$SS_L = \frac{0.36}{0.24} = 1.50$ $SS_R = \frac{0.09}{0.24} = 0.375$

Gain = 1.50 + 0.375 − 0 = 1.875

Split 2: cgpa <= 6.67

Yes → -0.6, 0.4 ; No → -0.6, 0.4, 0.4, 0.4

$SS_L = \frac{(-0.6+0.4)^2}{2*0.24}$ $SS_R = \frac{(-0.6+0.4+0.4+0.4)^2}{3*0.24}$

$SS_L = 0.08$ $SS_R = 0.05$

Gain = 0.08 + 0.05 − 0

Gain = 0.13

Split 3: cgpa <= 7.62

Yes → -0.6, 0.4, -0.6 ; No → 0.4, 0.4

$SS_L = \frac{(-0.6+0.4-0.6)^2}{0.24*3}$ $SS_R = \frac{(0.8)^2}{0.24*2}$

$SS_L = 0.875$ $SS_R = 1.33$

Gain = 0.87 + 1.33 − 0

Gain = 2.20

Split 4: cgpa <= 8.87

Yes → -0.6, 0.4, -0.6, 0.4 ; No → 0.4

$SS_L = \frac{(-0.6+0.4-0.6+0.4)^2}{4*0.24}$ $SS_R = \frac{(0.4)^2}{1*0.24}$

$SS_L = \frac{0.04}{0.24} = 0.16$ $SS_R = 0.66$

Gain = 0.16 + 0.66 − 0 = 0.82

The gain in similarity for condition cgpa <= 7.62 was the highest, so that can be chosen as the first point for making a split in the tree.

Now we need to divide this node again

cgpa	res1
5.70	-0.6
6.25	0.4
7.10	-0.6

Split 1: cgpa <= 5.97

Yes → -0.6 ; No → 0.4, -0.6

$SS_L = \frac{(-0.6)^2}{0.24}$ $SS_R = \frac{(0.4-0.6)^2}{2*0.24}$

$SS_L = 1.5$ $SS_R = 0.08$

Gain = 1.5 + 0.08 − 0.875

Gain = +0.705

Split 2: cgpa <= 6.67

Yes → -0.6, 0.4 ; No → -0.6

$SS_L = \frac{(-0.6+0.4)^2}{2*0.24}$ $SS_R = \frac{(-0.6)^2}{0.24}$

$SS_L = 0.08$ $SS_R = 1.5$

Gain = 0.08 + 1.5 − 0.875

Gain = 0.705

Since, gain for both splits are same we can choose any split for making the next node, our final tree looks like this:

cgpa <= 7.62

Yes → cgpa <= 5.97 ; No → 0.4, 0.4 (o/p = 1.66)

cgpa <= 5.97: Yes → -0.6 (o/p = -2.5) ; No → 0.4, -0.6 (o/p = -0.41)

Now, we need to calculate output for these leaves which is calculated using the formula:

$$\text{output} = \frac{\Sigma \text{residual}_i}{\Sigma \text{prevprob}(1-\text{prevprob})}$$

Now using this model, we predict the output for our second model (M_2) which is given as –

$fpred(M_2) = M_1 + 0.3 * (M_2)$

Cgpa	placed	M_1	Res 1	$Pred_2(M_2)$	$Prob(M_2)$	Res 2	
5.7	0	0.405	-0.6	-1.35	0.2	-0.2	$\Rightarrow -0.6+0.3*(-2.5) = -1.35$
6.25	1	0.405	0.4	0.277	0.57	0.43	$\Rightarrow 0.4+0.3*(-0.41) = 0.277$
7.1	0	0.405	-0.6	-0.723	0.33	-0.33	$\Rightarrow -0.6+0.3*(-.41) = -0.723$
8.15	1	0.405	0.4	0.71	0.71	0.29	$\Rightarrow 0.4+0.3*(1.66) = 0.71$
9.6	1	0.405	0.4	0.71	0.71	0.29	$\Rightarrow 0.4+0.3*(1.66) = 0.71$

For each log-odds we will calculate the probability using the formula – $p = 1/1+e^{-logodds}$

Finally, we calculate the residual by subtracting output (placed) column and probability (M_2) column, we can observe the residual gets decreased for most of the case as opposed to model M_1. Hence, we can add more models in this manner to reduce the residuals.