**The Math Behind XGBoost**

Building XGBoost from Scratch Using Python

[[Cristian Leo](https://medium.com/@cristianleo120?source=post_page-----3068c78aad9d--------------------------------)](https://medium.com/@cristianleo120?source=post_page-----3068c78aad9d--------------------------------)

[Cristian Leo](https://medium.com/@cristianleo120?source=post_page-----3068c78aad9d--------------------------------)

·

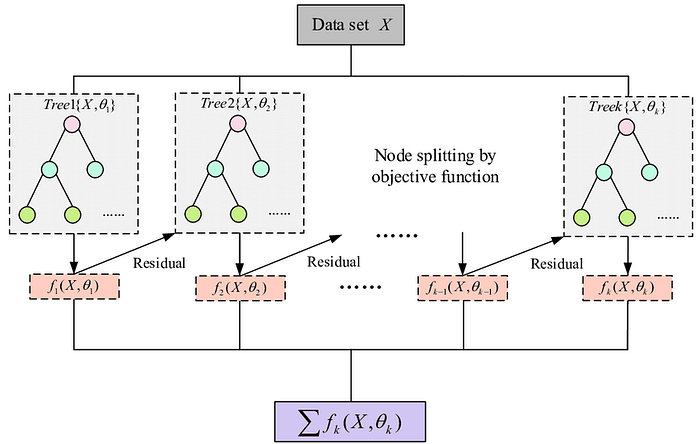
[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fc24a3d106811&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40cristianleo120%2Fthe-math-behind-xgboost-3068c78aad9d&user=Cristian+Leo&userId=c24a3d106811&source=post_page-c24a3d106811----3068c78aad9d---------------------post_header-----------)

17 min read

·

Jan 11

30



XGBoost Flow by [ResearchGate](https://www.researchgate.net/figure/Flow-chart-of-XGBoost_fig3_345327934)

**Introduction**

Gradient Boosting is a powerful and versatile machine learning technique that has gained immense popularity in solving complex predictive tasks. This method stands out for its effectiveness in handling a variety of data types, robustness against overfitting, and exceptional performance in both regression and classification tasks. The core idea behind Gradient Boosting is to sequentially build an ensemble of weak learners — typically decision trees — to create a model that outperforms any of the individual learners.

**Gradient Boosting Framework**

Let’s first outline the high level framework, and then dive deep in each of these steps:

**1. Initialization**

* **Start with a Base Model**: Typically, the model starts with a simple prediction for all instances, like the mean (for regression) or mode (for classification) of the target variable.
* **Initial Prediction**: This initial prediction serves as the starting point, and the algorithm will iteratively improve upon it.

**2. Iterative Improvement**

* **Sequentially Add Weak Learners**: The core idea is to add weak learners (usually decision trees) sequentially. A weak learner is a model that is slightly better than random guessing.
* **Residuals/Error Calculation**: After each tree is added, the algorithm calculates the residuals or errors (difference between the predicted and actual values).

**3. Gradient Descent Step**

* **Compute Negative Gradient**: In each iteration, the algorithm computes the negative gradient of the loss function with respect to the prediction. This gradient is the steepest descent, showing how the prediction should be changed to reduce the loss.
* **Fit New Model to Gradient**: A new weak learner (tree) is then fitted to these gradients.

**4. Update Model with Learning Rate**

* **Apply Learning Rate**: The predictions of the new tree are scaled by a parameter known as the learning rate (or shrinkage factor).
* **Update the Model**: The scaled predictions are added to the previous predictions to update the model. This learning rate controls how fast the model learns and helps in preventing overfitting.

**5. Regularization**

* **Control Model Complexity**: Various methods are used to regularize the model, like setting a maximum depth for trees, minimum samples for a leaf, and the number of trees (iterations).

**6. Stopping Criteria**

* **Determine When to Stop**: The algorithm stops adding trees either after a fixed number of trees are added or when the improvement drops below a threshold.

Now let’s dive in deep into each of the step.

**Initialization: Starting Model**

In the Gradient Boosting framework, the initialization step involves creating a starting model that provides a baseline prediction for all instances in the dataset. This initial model is typically very simple.

For regression tasks, the initial model often predicts the mean of the target variable, and for classification tasks, it might predict the mode (most frequent class) or the log odds ratio (in the case of logistic regression for binary classification).

**Regression**

In regression, the initial prediction ***F\_0(x)***can be the mean of the target variable ***y***. Mathematically, it is represented as:

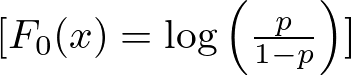


XGBoost’s regression formula

where:  
- ***N*** is the total number of instances in the training dataset.  
- ***y\_i***is the target value for the ***i***-th instance.  
- ***bar{y}***is the mean of all target values.

**Classification**

For binary classification, things are slightly more complex, as the initial model typically provides a probability estimate. The log odds of the positive class is a common initial prediction. The formula is:



XGBoost’s classification formula

where ***p***is the proportion of the positive class in the dataset. More simply, for a binary classification with classes ***0***and ***1***, ***p*** is the proportion of instances that belong to class ***1***.

For a multi-class classification problem, the initial model might predict the log odds for each class separately.

**Iterative Improvement**

The iterative improvement step in the Gradient Boosting framework is a crucial phase where the algorithm successively adds weak learners (usually decision trees) to the ensemble, specifically targeting the shortcomings of the existing combined model. Each new learner focuses on the errors or residuals made by the previous ensemble of learners.

Here’s a breakdown of the iterative improvement step:

**Calculate Residuals/Error**

For regression tasks, the residuals are typically the differences between the observed values and the current predictions. For classification, it involves calculating the derivative of the loss function with respect to the current model’s predictions.

Let’s denote:

* ***y\_i***as the true value for the \( i \)-th instance.
* ***F\_{m-1}(x\_i)*** as the current prediction for the ***i***-th instance after ***m-1*** iterations.

The residual for the ***i***-th instance at iteration ***m***is:



Residual Error

**Fit a Weak Learner to Residuals**

A new weak learner (e.g., a decision tree) is trained on the residuals. This learner aims to predict the residuals for each instance, essentially learning from the mistakes of the current model.

**Compute the Output of the Learner**

For regression, the output of the new learner is the predicted residual. In classification, it involves calculating a value that, when used to update the current model, will improve its accuracy.

Let’s denote the output of the new learner for the ***i***-th instance as ***h\_m(x\_i)***.

**Update the Model**

The model is updated by adding the scaled output of the new learner to the existing prediction.

The updated prediction for the ***i***-th instance is:



Where ***v*** is the learning rate, a value between 0 and 1 that scales the contribution of each tree. It helps in preventing overfitting and controls the rate at which the model learns.

**Repeat Steps**

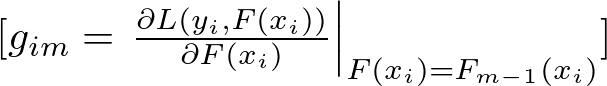
These steps are repeated for a specified number of iterations, or until a convergence criterion is met. Each iteration effectively reduces the residuals (in regression) or improves the accuracy (in classification) of the ensemble model.

**Gradient Descent Step**

The Gradient Descent step in Gradient Boosting is a crucial component that distinguishes it from other boosting methods. Instead of directly minimizing the residuals, as in traditional boosting methods, Gradient Boosting minimizes a loss function by moving in the direction of the steepest descent, as defined by the negative gradient. This step is iterative and occurs for each stage in the boosting process.

**1. Calculate the Gradient**

For each instance in the training set, compute the gradient of the loss function with respect to the model’s prediction.



Where:

* ***g\_{im}*** is the gradient for the ***i***-th instance at the ***m***-th iteration.
* ***F\_{m-1}(x\_i)***is the current prediction for the ***i***-th instance.

**2. Fit a Weak Learner to the Negative Gradients**

Train a new weak learner (e.g., a decision tree) to predict the negative gradients ***-g\_{im}***. This learner aims to move the prediction in the direction that most reduces the loss function.

**3. Determine the Step Size (Learning Rate)**

A line search is performed to determine the optimal step size, also known as the learning rate ***v***. This step controls how much we are adjusting the model in response to the estimated error each round.

**4. Update the Model**

The model is updated using the output from the new learner and the learning rate.



Where ***h\_m(x\_i)***is the prediction of the ***m***-th learner for the **i**-th instance.

**Update Model with Learning Rate**

The “Update the Model with Learning Rate” step is a critical phase in the Gradient Boosting framework. Here, the predictions from the newly added weak learner are incorporated into the existing model. The learning rate, a key hyperparameter in this process, controls how these new predictions are weighted, thereby impacting the overall model update.

**1. Apply the Learning Rate**

The learning rate is a factor ***v*** in the range (0, 1] that scales the contribution of each weak learner. It’s also known as the shrinkage factor.  
It slows down the learning process, allowing for more robust models by giving more opportunity for future trees to learn from the residuals left by previous trees.

**2. Combine the Weak Learner’s Predictions with the Current Model**

The model is updated by adding the output of the new weak learner, multiplied by the learning rate, to the previous model.

The updated model after adding the ***m***-th tree is given by:



Here, ***h\_m(x)*** represents the predictions of the ***m***-th weak learner (e.g., a decision tree).

**Regularization**

Regularization in Gradient Boosting is a crucial mechanism to prevent overfitting, ensuring that the model generalizes well to unseen data. It involves adding constraints or penalties to the model to control its complexity, making it less sensitive to the noise in the training data.

**Tree Constraints**

* **Max Depth**: Limiting the maximum depth of each tree. Deeper trees can capture more complex patterns but might overfit.
* **Min Samples per Leaf**: Setting a minimum number of samples that are required in a leaf node. This prevents the model from learning rules that are too specific to the training data.
* **Max Leaves**: Restricting the maximum number of leaf nodes in a tree.

**Shrinkage (Learning Rate)**

As discussed earlier, using a learning rate less than 1 reduces the contribution of each tree and slows down learning, which can lead to better generalization.

**Subsampling of Data (Stochastic Gradient Boosting)**

* **Feature Subsampling**: Using only a random subset of features for fitting each tree, similar to the Random Forest approach.
* **Data Subsampling**: Using a random subset of the training data to fit each tree. This technique is also known as “bagging”.

**Penalty on Leaf Weights (L1/L2 Regularization)**

* **L1 (Lasso) Regularization**: Adds a penalty equal to the absolute value of the magnitude of the coefficients. This can lead to sparse models where some feature weights are exactly zero.
* **L2 (Ridge) Regularization**: Adds a penalty equal to the square of the magnitude of the coefficients. This discourages large weights but typically does not result in zero weights.

**Mathematical Representation**

For a given loss function \( L \), the regularized objective to minimize can be represented as:



Where:

* ***L(y, F(x))***is the loss function that measures the fit of the model to the training data.
* ***R(F)*** is the regularization term (L1, L2, or both).
* ***lambda*** is the regularization coefficient that controls the trade-off between the loss and the regularization term.

**Stopping Criteria**

The stopping criteria in Gradient Boosting are rules or thresholds used to decide when the training of the model should be stopped. These criteria are essential to prevent overfitting and to ensure computational efficiency. Properly set stopping criteria help in determining the optimal number of boosting iterations, which is a key factor in the performance of Gradient Boosting models.

**Number of Trees (Boosting Rounds)**

Often, a maximum number of trees (boosting rounds) is set as a stopping criterion. Training stops once this number is reached.  
This approach is straightforward but may require tuning to find the optimal number of trees.

**Performance Threshold**

* **Validation-Based Stopping**: The training stops if the model’s performance on a validation set does not improve for a specified number of consecutive trees.
* **Early Stopping**: For example, if the model’s accuracy on a validation set does not improve for 10 consecutive boosting rounds, training is halted. This is a form of early stopping.

**Minimal Loss Reduction**

Set a threshold for the minimum reduction in loss required to continue training. If adding new trees does not decrease the loss beyond this threshold, training stops.  
This criterion ensures that training only continues while it is yielding significant improvements in model performance.

**Change in Prediction**

Training can also be stopped if the change in predictions between consecutive iterations falls below a certain threshold, indicating that additional trees are not significantly altering the model.

**Benefits**

XGBoost offers a number of benefits, including:

* **Accuracy**: XGBoost is typically very accurate machine learning model.
* **Speed**: XGBoost is one of the fastest gradient boosting algorithms.
* **Scalability**: XGBoost can handle large datasets with millions of data points.
* **Flexibility**: XGBoost can be used for both classification and regression tasks.

**Pitfalls**

XGBoost also has some pitfalls, including:

* **Overfitting**: XGBoost is prone to overfitting, especially if the number of trees is too large.
* **Hyperparameter tuning:** XGBoost has a number of hyperparameters that need to be tuned to achieve optimal performance.

**Implementation from scratch in Python**

Full Python Code: <https://github.com/Ekeany/XGBoost-From-Scratch/blob/master/XGBoost.py>

Notebook with practical implementation: <https://github.com/cristianleoo/models-from-scratch-python/blob/main/xgboost.ipynb>

Here is a simple implementation of XGBoost from scratch in Python:

class Node:  
 """  
 A node class for a decision tree.  
 """  
 def \_\_init\_\_(self, x, gradient, hessian, idxs, subsample\_cols=1 , min\_leaf=5, min\_child\_weight=1 ,depth=10, lambda\_=1, gamma=1, eps=0.1):  
 """  
 Constructor to initialize the node with data and parameters.  
 Parameters:  
 - x: Input data for the node.  
 - gradient: Gradient information for gradient boosting.  
 - hessian: Hessian information for second-order optimization.  
 - idxs: Indices of the data points in the node.  
 - subsample\_cols: Fraction of columns to consider for splitting.  
 - min\_leaf: Minimum number of samples required in a leaf node.  
 - min\_child\_weight: Minimum sum of instance weight(hessian) needed in a child.  
 - depth: Maximum depth of the tree.  
 - lambda\_: Regularization parameter.  
 - gamma: Minimum loss reduction required to make a further partition.  
 - eps: Epsilon value for quantile sketch method.  
 """  
 self.x, self.gradient, self.hessian = x, gradient, hessian  
 self.idxs = idxs   
 self.depth = depth  
 self.min\_leaf = min\_leaf  
 self.lambda\_ = lambda\_  
 self.gamma = gamma  
 self.min\_child\_weight = min\_child\_weight  
 self.row\_count = len(idxs)  
 self.col\_count = x.shape[1]  
 self.subsample\_cols = subsample\_cols  
 self.eps = eps  
 self.column\_subsample = np.random.permutation(self.col\_count)[:round(self.subsample\_cols\*self.col\_count)]  
 self.val = self.compute\_gamma(self.gradient[self.idxs], self.hessian[self.idxs])  
 self.score = float('-inf')  
 self.find\_varsplit()  
   
 def compute\_gamma(self, gradient, hessian):  
 """  
 Computes the gamma value for the node.  
 Gamma is calculated as negative sum of gradient divided by the sum of hessian plus lambda.  
   
 Parameters:  
 - gradient: Gradient information for gradient boosting.  
 - hessian: Hessian information for second-order optimization.  
   
 Returns:  
 - gamma: Gamma value for the node.  
 """  
 return(-np.sum(gradient)/(np.sum(hessian) + self.lambda\_))  
   
 def find\_varsplit(self):  
 """  
 Identifies the best variable to split on.  
 Iterates through the subset of columns and finds the best greedy split.  
 """  
 for c in self.column\_subsample: self.find\_greedy\_split(c)  
 if self.is\_leaf: return  
 x = self.split\_col  
 lhs = np.nonzero(x <= self.split)[0]  
 rhs = np.nonzero(x > self.split)[0]  
 self.lhs = Node(x = self.x, gradient = self.gradient, hessian = self.hessian, idxs = self.idxs[lhs], min\_leaf = self.min\_leaf, depth = self.depth-1, lambda\_ = self.lambda\_ , gamma = self.gamma, min\_child\_weight = self.min\_child\_weight, eps = self.eps, subsample\_cols = self.subsample\_cols)  
 self.rhs = Node(x = self.x, gradient = self.gradient, hessian = self.hessian, idxs = self.idxs[rhs], min\_leaf = self.min\_leaf, depth = self.depth-1, lambda\_ = self.lambda\_ , gamma = self.gamma, min\_child\_weight = self.min\_child\_weight, eps = self.eps, subsample\_cols = self.subsample\_cols)  
   
 def find\_greedy\_split(self, var\_idx):  
 """  
 Finds the best split point for a given variable using a greedy approach.  
 Iterates through each row and evaluates potential splits.  
  
 Parameters:  
 - var\_idx: Index of the variable to split on.  
 """  
 x = self.x[self.idxs, var\_idx]  
 for r in range(self.row\_count):  
 lhs = x <= x[r]  
 rhs = x > x[r]  
 lhs\_indices = np.nonzero(lhs)[0]  
 rhs\_indices = np.nonzero(rhs)[0]  
 lhs\_sum = self.hessian[lhs\_indices].sum()  
 rhs\_sum = self.hessian[rhs\_indices].sum()  
 # Ensures minimum leaf size and child weight before considering the split.  
 if(rhs.sum() < self.min\_leaf or lhs.sum() < self.min\_leaf   
 or lhs\_sum < self.min\_child\_weight  
 or rhs\_sum < self.min\_child\_weight): continue  
 curr\_score = self.gain(lhs, rhs)  
 # Updates the best split if a better score is found.  
 if curr\_score > self.score:   
 self.var\_idx = var\_idx  
 self.score = curr\_score  
 self.split = x[r]  
   
 def weighted\_qauntile\_sketch(self, var\_idx):  
 """  
 Finds the best split point for a given variable using a weighted quantile sketch approach.  
 Iterates through each row and evaluates potential splits.  
  
 Parameters:  
 - var\_idx: Index of the variable to split on.  
 """  
 x = self.x[self.idxs, var\_idx]  
 hessian\_ = self.hessian[self.idxs]  
 df = pd.DataFrame({'feature':x,'hess':hessian\_})  
 df.sort\_values(by=['feature'], ascending = True, inplace = True)  
 hess\_sum = df['hess'].sum()   
 df['rank'] = df.apply(lambda x : (1/hess\_sum)\*sum(df[df['feature'] < x['feature']]['hess']), axis=1)  
   
 for row in range(df.shape[0]-1):  
 # look at the current rank and the next ran  
 rk\_sk\_j, rk\_sk\_j\_1 = df['rank'].iloc[row:row+2]  
 diff = abs(rk\_sk\_j - rk\_sk\_j\_1)  
 if(diff >= self.eps):  
 continue  
   
 split\_value = (df['rank'].iloc[row+1] + df['rank'].iloc[row])/2  
 lhs = x <= split\_value  
 rhs = x > split\_value  
   
 lhs\_indices = np.nonzero(x <= split\_value)[0]  
 rhs\_indices = np.nonzero(x > split\_value)[0]  
 if(rhs.sum() < self.min\_leaf or lhs.sum() < self.min\_leaf   
 or self.hessian[lhs\_indices].sum() < self.min\_child\_weight  
 or self.hessian[rhs\_indices].sum() < self.min\_child\_weight): continue  
   
 curr\_score = self.gain(lhs, rhs)  
 if curr\_score > self.score:   
 self.var\_idx = var\_idx  
 self.score = curr\_score  
 self.split = split\_value  
   
 def gain(self, lhs, rhs):  
 """  
 Computes the gain in loss function for a given split.  
  
 Parameters:  
 - lhs: Left hand side of the split.  
 - rhs: Right hand side of the split.  
  
 Returns:  
 - gain: Gain in loss function for the split.  
 """  
 gradient = self.gradient[self.idxs]  
 hessian = self.hessian[self.idxs]  
 lhs\_gradient = gradient[lhs].sum()  
 lhs\_hessian = hessian[lhs].sum()  
 rhs\_gradient = gradient[rhs].sum()  
 rhs\_hessian = hessian[rhs].sum()  
 total\_gradient = lhs\_gradient + rhs\_gradient  
 total\_hessian = lhs\_hessian + rhs\_hessian  
   
 gain = 0.5 \*( (lhs\_gradient\*\*2/(lhs\_hessian + self.lambda\_)) + (rhs\_gradient\*\*2/(rhs\_hessian + self.lambda\_)) - (total\_gradient\*\*2/(total\_hessian + self.lambda\_))) - self.gamma  
 return(gain)  
   
 @property  
 def split\_col(self):  
 """  
 Returns the column of the split variable.  
 """  
 return self.x[self.idxs , self.var\_idx]  
   
 @property  
 def is\_leaf(self):  
 """  
 Returns True if the node is a leaf node.  
 """  
 return self.score == float('-inf') or self.depth <= 0   
  
 def predict(self, x):  
 """  
 Predicts the value for a given input.  
  
 Parameters:  
 - x: Input data.  
  
 Returns:  
 - np.array: Predicted values.  
 """  
 return np.array([self.predict\_row(xi) for xi in x])  
   
 def predict\_row(self, xi):  
 """  
 Predicts the value for a given input row.  
  
 Parameters:  
 - xi: Input row.  
  
 Returns:  
 - np.array: Predicted value.  
 """  
 if self.is\_leaf:  
 return(self.val)  
  
 node = self.lhs if xi[self.var\_idx] <= self.split else self.rhs  
 return node.predict\_row(xi)

The Node class represents a single node in a decision tree, specifically designed for use in gradient boosting frameworks. The class provides methods for initializing the node, splitting the data, and making predictions. Here's an explanation of the class and its methods:

\_\_init\_\_: The constructor initializes the node with the given data and parameters. It sets up the basic structure required for subsequent operations like finding the best split.

Parameters:

* x: The features of the dataset.
* gradient: Gradient information used in gradient boosting.
* hessian: Hessian information for second-order optimization.
* idxs: Indices of the data points that this node is responsible for.
* subsample\_cols: Proportion of columns to consider for finding the best split.
* min\_leaf: Minimum number of samples required in a leaf node to avoid overfitting.
* min\_child\_weight: Minimum sum of instance weights (hessian) required in a child node.
* depth: Maximum depth of the tree, used to control overfitting.
* lambda\_: Regularization parameter to avoid overfitting.
* gamma: Minimum loss reduction required to make a further partition on a leaf node.
* eps: Epsilon value for the weighted quantile sketch method.

compute\_gamma: This method computes the gamma value for the node, which is used in the calculation of the splitting criteria.

find\_varsplit: This method finds the best variable to split on. It iterates over a subset of columns to identify the most effective split based on the criteria defined by the algorithm.

find\_greedy\_split: For a given variable, this method attempts to find the best split point using a greedy approach. It evaluates the effectiveness of each possible split based on a defined score.

weighted\_qauntile\_sketch: This method finds the best split point using a weighted quantile sketch approach. It's an efficient way to find a good split point, especially for large datasets.

gain: Calculates the gain in the loss function for a given split. This is used to evaluate how good a particular split is.

split\_col (property): Returns the column of the data that the node splits on.

is\_leaf (property): Determines whether the node is a leaf node. A node is a leaf if it cannot be split further.

predict: Makes predictions for a given set of inputs. It applies the decision tree rules to each input row.

predict\_row: Makes a prediction for a single row of input. If the node is a leaf, it returns the value of the node. Otherwise, it passes the input to the appropriate child node.

class XGBoostTree:  
 def fit(self, x, gradient, hessian, subsample\_cols = 0.8 , min\_leaf = 5, min\_child\_weight = 1 ,depth = 10, lambda\_ = 1, gamma = 1, eps = 0.1):  
 """  
 Fits a decision tree to the data.  
  
 Parameters:  
 - x: Input data.  
 - gradient: Gradient information for gradient boosting.  
 - hessian: Hessian information for second-order optimization.  
 - subsample\_cols: Fraction of columns to consider for splitting.  
 - min\_leaf: Minimum number of samples required in a leaf node.  
 - min\_child\_weight: Minimum sum of instance weight(hessian) needed in a child.  
 - depth: Maximum depth of the tree.  
 - lambda\_: Regularization parameter.  
 - gamma: Minimum loss reduction required to make a further partition.  
 - eps: Epsilon value for quantile sketch method.  
  
 Returns:  
 - self: Trained decision tree.  
 """  
 self.dtree = Node(x, gradient, hessian, np.array(np.arange(len(x))), subsample\_cols, min\_leaf, min\_child\_weight, depth, lambda\_, gamma, eps)  
 return self  
   
 def predict(self, X):  
 """  
 Predicts the value for a given input.  
  
 Parameters:  
 - X: Input data.  
  
 Returns:  
 - np.array: Predicted values.  
 """  
 return self.dtree.predict(X)

fit Method: This method is used to fit the decision tree model to the given data. Functionality: The method initializes a Node instance (presumably defined elsewhere, as in your previous code) with the provided parameters, effectively starting the construction of the decision tree. Returns the XGBoostTree instance itself (self), now containing the trained decision tree (self.dtree).

predict Method: This method is used for making predictions on new data using the trained decision tree. It invokes the predict method of the Node class on the input data. This process traverses the decision tree according to the values in each row of X and uses the learned rules from the training process to make predictions. Returns an array of predicted values corresponding to the input data.

class XGBoostClassifier:  
 def \_\_init\_\_(self, random\_state=None):  
 self.estimators = []  
 self.random\_state = random\_state  
 np.random.seed(self.random\_state)  
   
 @staticmethod  
 def sigmoid(x):  
 """  
 Computes the sigmoid function.  
   
 Parameters:  
 - x: Input data.  
   
 Returns:  
 - np.array: Sigmoid of the input data.  
 """  
 return 1 / (1 + np.exp(-x))  
   
 def grad(self, preds, labels):  
 """  
 Computes the gradient of the log loss function.  
   
 Parameters:  
 - preds: Predicted values.  
 - labels: Actual values.  
   
 Returns:  
 - np.array: Gradient of the log loss function.  
 """  
 preds = self.sigmoid(preds)  
 return(preds - labels)  
   
 def hess(self, preds, labels):  
 """  
 Computes the hessian of the log loss function.  
   
 Parameters:  
 - preds: Predicted values.  
   
 Returns:  
 - np.array: Hessian of the log loss function.  
 """  
 preds = self.sigmoid(preds)  
 return(preds \* (1 - preds))  
   
 @staticmethod  
 def log\_odds(column):  
 """  
 Computes the log odds of a binary variable.  
  
 Parameters:  
 - column: Binary variable.  
  
 Returns:  
 - np.array: Log odds of the binary variable.  
 """  
 binary\_yes = np.count\_nonzero(column == 1)  
 binary\_no = np.count\_nonzero(column == 0)  
 return(np.log(binary\_yes/binary\_no))  
   
   
 def fit(self, X, y, subsample\_cols=1 , min\_child\_weight=1, depth=5, min\_leaf=5, learning\_rate=0.1, boosting\_rounds=10, lambda\_=1.5, gamma=1, eps=0.1):  
 """  
 Fits a gradient boosted decision tree to the data.  
  
 Parameters:  
 - X: Input data.  
 - y: Target variable.  
 - subsample\_cols: Fraction of columns to consider for splitting.  
 - min\_leaf: Minimum number of samples required in a leaf node.  
 - min\_child\_weight: Minimum sum of instance weight(hessian) needed in a child.  
 - depth: Maximum depth of the tree.  
 - lambda\_: Regularization parameter.  
 - gamma: Minimum loss reduction required to make a further partition.  
 - eps: Epsilon value for quantile sketch method.  
 - learning\_rate: Learning rate for gradient boosting.  
 - boosting\_rounds: Number of boosting rounds.  
  
 Returns:  
 - self: Trained gradient boosted decision tree.  
 """  
 self.X, self.y = X, y  
 self.depth = depth  
 self.subsample\_cols = subsample\_cols  
 self.eps = eps  
 self.min\_child\_weight = min\_child\_weight   
 self.min\_leaf = min\_leaf  
 self.learning\_rate = learning\_rate  
 self.boosting\_rounds = boosting\_rounds   
 self.lambda\_ = lambda\_  
 self.gamma = gamma  
 self.base\_pred = np.full((X.shape[0], 1), 1).flatten().astype('float64')  
 Grad = self.grad(self.base\_pred, self.y)  
 Hess = self.hess(self.base\_pred, self.y)  
 for \_ in range(self.boosting\_rounds):  
 boosting\_tree = XGBoostTree().fit(self.X, Grad, Hess, depth = self.depth, min\_leaf = self.min\_leaf, lambda\_ = self.lambda\_, gamma = self.gamma, eps = self.eps, min\_child\_weight = self.min\_child\_weight, subsample\_cols = self.subsample\_cols)  
 self.base\_pred += self.learning\_rate \* boosting\_tree.predict(self.X)  
 self.estimators.append(boosting\_tree)  
   
 def predict\_proba(self, X):  
 """  
 Predicts the probability of the positive class for a given input.  
  
 Parameters:  
 - X: Input data.  
  
 Returns:  
 - np.array: Predicted probabilities.  
 """  
 pred = np.zeros(X.shape[0])  
   
 for estimator in self.estimators:  
 pred += self.learning\_rate \* estimator.predict(X)   
   
 return self.sigmoid(np.full((X.shape[0], 1), 1).flatten().astype('float64') + pred)  
   
 def predict(self, X):  
 pred = np.zeros(X.shape[0])  
 for estimator in self.estimators:  
 pred += self.learning\_rate \* estimator.predict(X)   
   
 predicted\_probas = self.sigmoid(np.full((X.shape[0], 1), 1).flatten().astype('float64') + pred)  
 preds = np.where(predicted\_probas > np.mean(predicted\_probas), 1, 0)  
 return preds

The XGBoostClassifier encapsulates the logic for training a series of decision trees in a boosting manner and making predictions based on their collective output. It uses gradient and Hessian information to guide the learning process and employs the sigmoid function to translate raw model outputs into probabilities. The boosting process involves iteratively correcting the errors of previous trees, making the model more robust with each round. This implementation includes detailed control over the training process, allowing for fine-tuning to specific datasets and requirements.

sigmoid Static Method: Computes the sigmoid function, which is essential for transforming outputs into probabilities.

grad Method: Computes the gradient of the log loss function, necessary for the gradient boosting process.

hess Method: Calculates the Hessian (second-order derivative) of the log loss function. It’s used in the optimization process of the model.

log\_odds Static Method: Computes the log odds of a binary variable, useful for initializing predictions.

fit Method: Trains the gradient boosted decision tree model on the given data. Initializes base predictions and computes initial gradient and Hessian. Iteratively builds decision trees (using XGBoostTree) that learn from the residuals of the previous trees.

predict\_proba Method: Predicts the probability of the positive class for given inputs.

predict Method:Makes final predictions based on the combined output of all the boosted trees.

Consider that this is a simple implementation of XGBoost for educational purposes, and it does not include all of the features that are typically found in an XGBoost implementation.

**Conclusion**

XGBoost’s blend of power and practicality makes it an indispensable algorithm for anyone looking to delve into the world of machine learning. As we continue to advance in data science and machine learning, techniques like XGBoost will undoubtedly play a pivotal role in driving innovation and solving real-world problems.

[Xgboost](https://medium.com/tag/xgboost?source=post_page-----3068c78aad9d---------------xgboost-----------------)

[Machine Learning](https://medium.com/tag/machine-learning?source=post_page-----3068c78aad9d---------------machine_learning-----------------)

[AI](https://medium.com/tag/ai?source=post_page-----3068c78aad9d---------------ai-----------------)

[Python](https://medium.com/tag/python?source=post_page-----3068c78aad9d---------------python-----------------)

[Data Science](https://medium.com/tag/data-science?source=post_page-----3068c78aad9d---------------data_science-----------------)