**What category of algorithms does XGBoost belong to?**

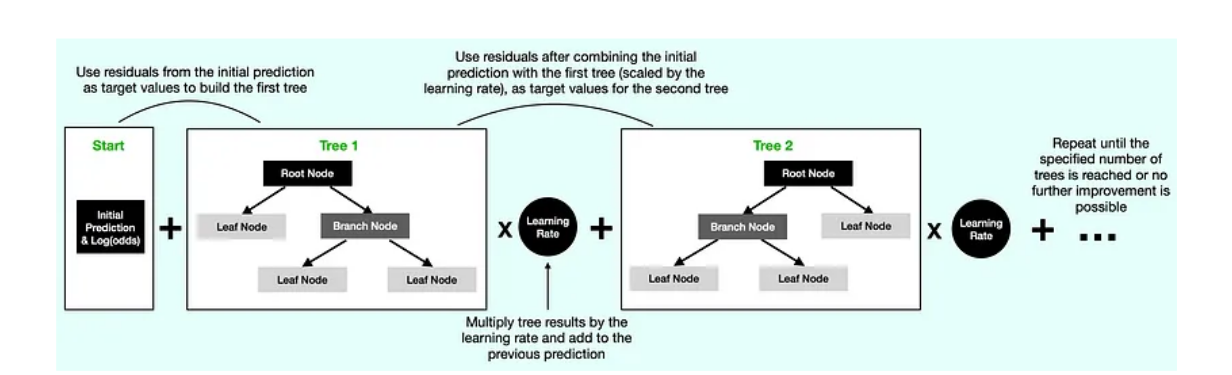
Extreme Gradient Boosting is a tree-based algorithm, which sits under the supervised branch of Machine Learning. It can be used for both classification and regression problems.

# XGBoost is a more refined and optimized version of the regular Gradient Boosting algorithm. In most cases, the results produced by these algorithms may end up being very similar. If you work with huge datasets, then Extreme Gradient Boosting should be a better choice for you.

# Basics of XGBoost and Gradient Boosting

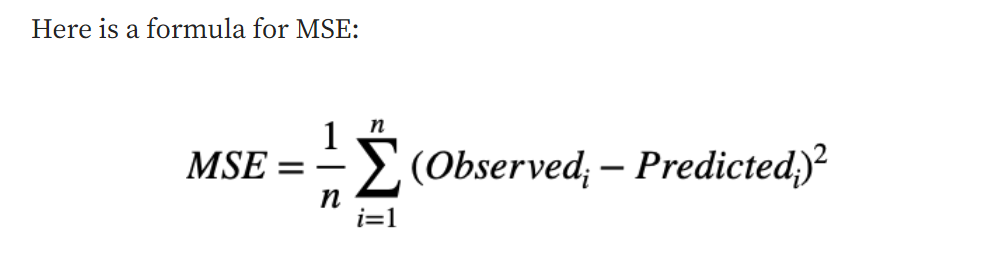
Before diving into the details, let’s review the basics of these algorithms.

* **Tree-based algorithms**— both XGBoost and Gradient Boosting use decision trees as their base estimators.
* **Prediction target**— the trees are built using residuals, not the actual class labels. Hence, despite us focusing on classification problems, the base estimators in these algorithms are regression trees and not classification trees. This is because residuals are continuous and not discrete. At the same time, though, some of the formulas you will see below are unique for classification, so please don’t assume that exactly the same applies to regression problems.
* **Tree depth** — both algorithms allow you to control the maximum size of the trees to minimize the risk of overfitting the data.
* **Ensemble methods**— similar to Random Forest or AdaBoost, these algorithms build many trees in the process. In the end, the final prediction is based on all of the trees.
* **Learning rate** — the value of each tree is scaled by the learning rate. This enables the algorithm to have a more gradual and steady improvement at each step.
* **Process map** — finally, here is a simple illustration of the process used by Gradient Boosting and XGBoost.



## Gradient Boosting

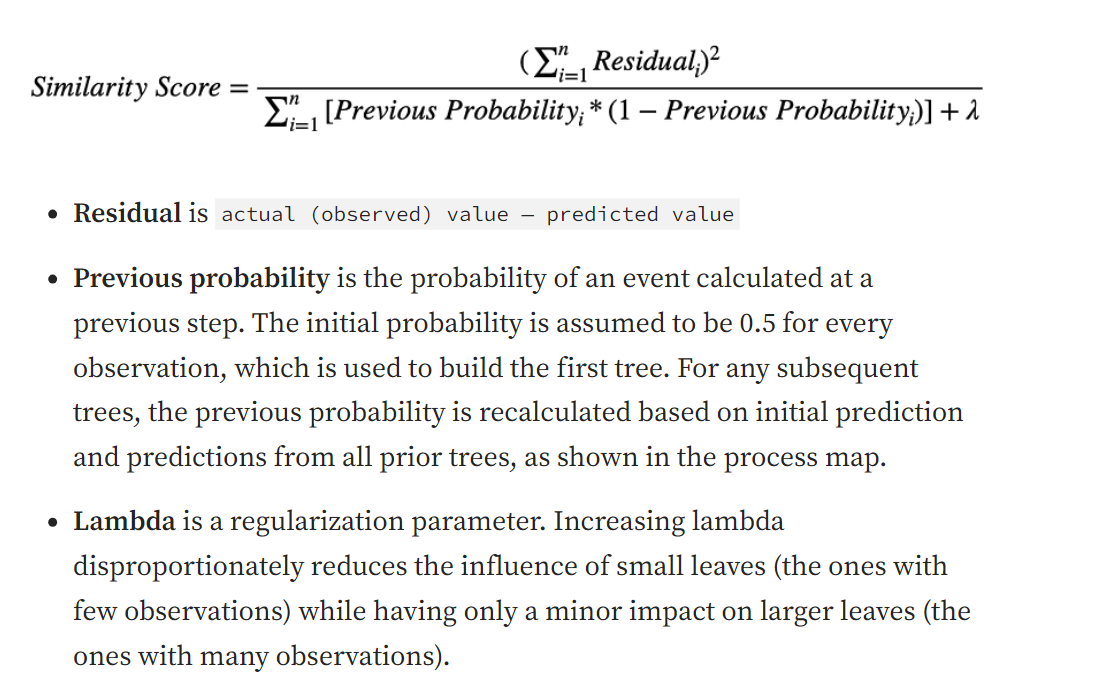
You will be happy to hear that regular Gradient Boosting uses a standard method to build regression trees, where a typical metric such as MSE (Mean Squared Error) or a similar one is used to determine the best split for the tree.

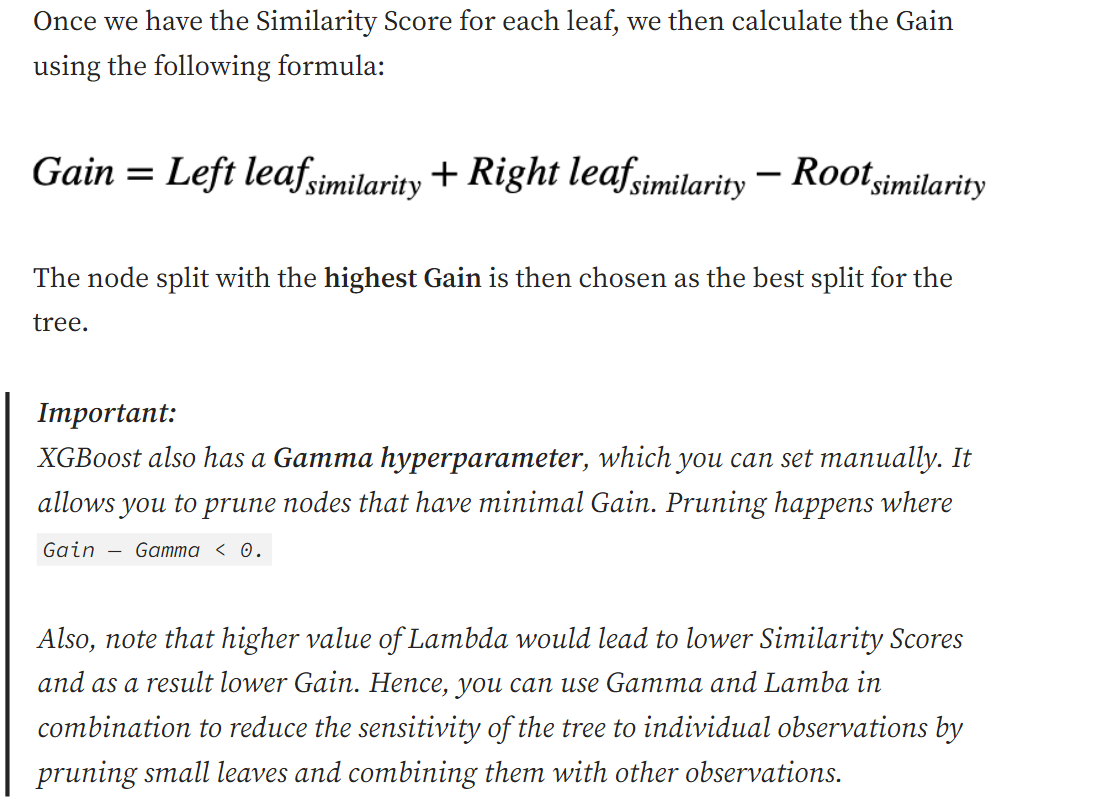


The algorithm calculates MSE for every possible node split and then picks the one with the lowest MSE as the one to use in the tree.

## **XGBoost**

Contrary to regular Gradient Boosting, XGBoost uses its own method of building trees where the Similarity Score and Gain determine the best node splits.

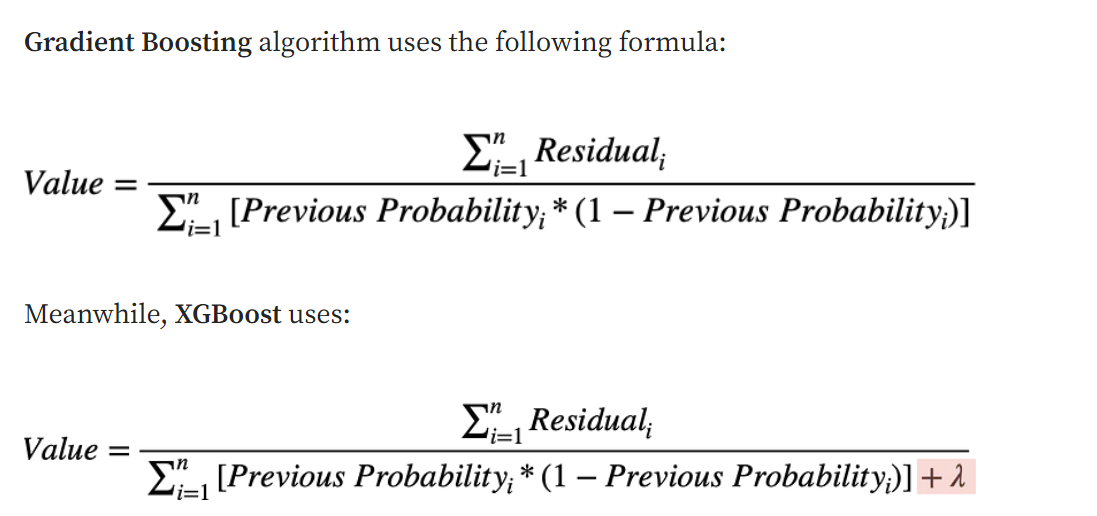




**Pruning** is a [data compression](https://en.wikipedia.org/wiki/Data_compression) technique in [machine learning](https://en.wikipedia.org/wiki/Machine_learning) and [search algorithms](https://en.wikipedia.org/wiki/Search_algorithm) that reduces the size of [decision trees](https://en.wikipedia.org/wiki/Decision_tree_learning) by removing sections of the tree that are non-critical and redundant to classify instances. Pruning reduces the complexity of the final [classifier](https://en.wikipedia.org/wiki/Statistical_classification), and hence improves predictive accuracy by the reduction of [overfitting](https://en.wikipedia.org/wiki/Overfitting).

# Calculating output values

The way the output value for each leaf is calculated is almost identical between the two algorithms, with the only difference being the lambda hyperparameter.



# XGBoost optimizations

In addition to using its own way of building and pruning trees, XGBoost also has several optimizations built-in to make the training faster when working with huge datasets. Here are a few of the main ones:

* **Approximate Greedy Algorithm**— uses weighted quantiles when looking for the best node split instead of evaluating every possible split.
* **Parallel Learning**— it can split up the data into smaller datasets to run processes in parallel.
* **Sparsity-Aware Split Finding** — when you have some missing data, it calculates Gain by putting observations with missing values into the left leaf. It then does the same by putting them into the right leaf and chooses the scenario that produces higher Gain.
* **Cash-Aware Access**— XGBoost uses the CPU’s cache memory to store gradients so it can calculate similarity scores faster.

## **Booster Parameters**

Though there are 2 types of boosters, I’ll consider only **tree booster** here because it always outperforms the linear booster, and thus the latter is rarely used.

1. **eta [default=0.3]**
   * Analogous to the learning rate in GBM
   * Makes the model more robust by shrinking the weights on each step
   * Typical final values to be used: 0.01-0.2
2. **min\_child\_weight [default=1]**
   * Defines the minimum sum of weights of all observations required in a child.
   * This is similar to **min\_child\_leaf** in GBM but not exactly. This refers to the min “sum of weights” of observations, while GBM has the min “number of observations”.
   * Used to control over-fitting. Higher values prevent a model from learning relations that might be highly specific to the particular sample selected for a tree.
   * Too high values can lead to under-fitting; hence, it should be tuned using CV.
3. **max\_depth [default=6]**
   * The maximum depth of a tree is the same as GBM.
   * Used to control over-fitting as higher depth will allow the model to learn relations very specific to a particular sample.
   * It should be tuned using CV.
   * Typical values: 3-10
4. **max\_leaf\_nodes**
   * The maximum number of terminal nodes or leaves in a tree.
   * It can be defined in place of max\_depth. Since binary trees are created, a depth of ‘n’ would produce a maximum of 2^n leaves.
   * If this is defined, GBM will ignore max\_depth.
5. **gamma [default=0]**
   * A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.
   * Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
6. **max\_delta\_step [default=0]**
   * In the maximum delta step, we allow each tree’s weight estimation to be. If the value is set to 0, there is no constraint. If it is set to a positive value, it can help make the update step more conservative.
   * Usually, this parameter is not needed, but it might help in logistic regression when the class is extremely imbalanced.
   * This is generally not used, but you can explore further if you wish.
7. **subsample [default=1]**
   * Same as the subsample of GBM. Denotes the fraction of observations to be random samples for each tree.
   * Lower values make the algorithm more conservative and prevent overfitting, but too small values might lead to under-fitting.
   * Typical values: 0.5-1
8. **colsample\_bytree [default=1]**
   * Similar to max\_features in GBM. Denotes the fraction of columns to be random samples for each tree.
   * Typical values: 0.5-1
9. **colsample\_bylevel [default=1]**
   * Denotes the subsample ratio of columns for each split in each level.
   * I don’t use this often because subsample and colsample\_bytree will do your job. but you can explore further if you feel so.
10. **lambda [default=1]**
    * L2 regularization term on weights (analogous to Ridge regression)
    * This is used to handle the regularization part of XGBoost. Though many data scientists don’t use it often, it should be explored to reduce overfitting.
11. **alpha [default=0]**
    * L1 regularization term on weight (analogous to Lasso regression)
    * It can be used in case of very high dimensionality so that the algorithm runs faster when implemented
12. **scale\_pos\_weight [default=1]**
    * A value greater than 0 should be used in case of high-class imbalance as it helps in faster convergence.

# XGBoost optimizations

In addition to using its own way of building and pruning trees, XGBoost also has several optimizations built-in to make the training faster when working with huge datasets. Here are a few of the main ones:

* **Approximate Greedy Algorithm**— uses weighted quantiles when looking for the best node split instead of evaluating every possible split.
* **Parallel Learning**— it can split up the data into smaller datasets to run processes in parallel.
* **Sparsity-Aware Split Finding** — when you have some missing data, it calculates Gain by putting observations with missing values into the left leaf. It then does the same by putting them into the right leaf and chooses the scenario that produces higher Gain.
* **Cash-Aware Access**— XGBoost uses the CPU’s cache memory to store gradients so it can calculate similarity scores faster.

### **Disadvantages of XGBoost:**

1. Computational Complexity: XGBoost can be computationally intensive, especially when training large models, making it less suitable for resource-constrained systems.
2. Overfitting: XGBoost can be prone to overfitting, especially when trained on small datasets or when too many trees are used in the model.
3. Hyperparameter Tuning: XGBoost has many hyperparameters that can be adjusted, making it important to properly tune the parameters to optimize performance. However, finding the optimal set of parameters can be time-consuming and requires expertise.
4. Memory Requirements: XGBoost can be memory-intensive, especially when working with large datasets, making it less suitable for systems with limited memory resources.