**XGBoost**

**XGBoost** is an optimized distributed gradient boosting library designed for efficient and scalable training of machine learning models. It is an ensemble learning method that combines the predictions of multiple weak models to produce a stronger prediction. XGBoost stands for “Extreme Gradient Boosting” and it has become one of the most popular and widely used machine learning algorithms due to its ability to handle large datasets and its ability to achieve state-of-the-art performance in many machine learning tasks such as classification and regression.

One of the key features of XGBoost is its efficient handling of missing values, which allows it to handle real-world data with missing values without requiring significant pre-processing. Additionally, XGBoost has built-in support for parallel processing, making it possible to train models on large datasets in a reasonable amount of time.

XGBoost can be used in a variety of applications, including Kaggle competitions, recommendation systems, and click-through rate prediction, among others. It is also highly customizable and allows for fine-tuning of various model parameters to optimize performance.

XgBoost stands for Extreme Gradient Boosting, which was proposed by the researchers at the University of Washington. It is a library written in C++ which optimizes the training for Gradient Boosting.

Before understanding the XGBoost, we first need to understand the trees especially the decision tree**:**

**Decision Tree:**

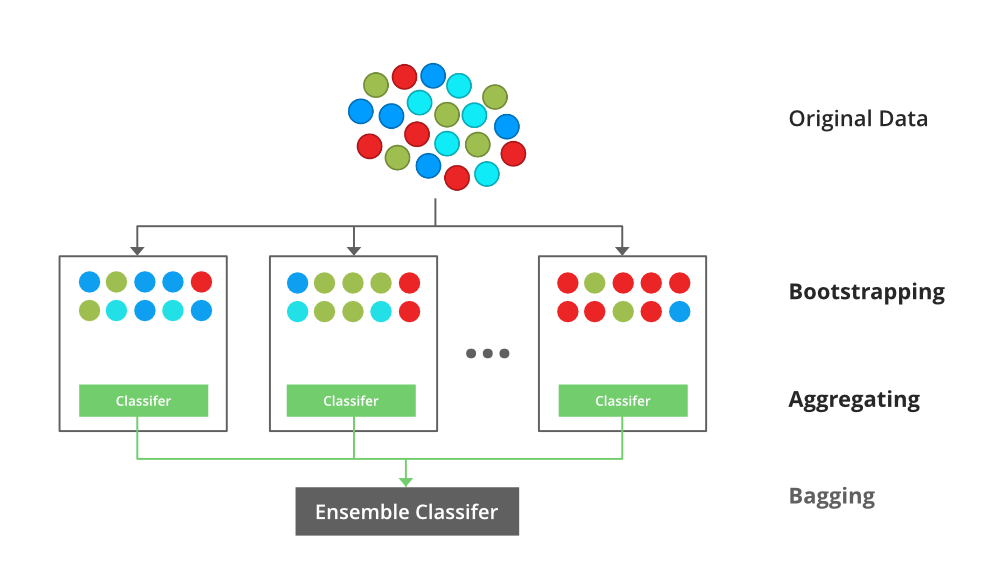
A Decision tree is a flowchart-like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.

A tree can be *“learned”* by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner calledrecursive partitioning. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions.

**Bagging:**

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.  
Each base classifier is trained in parallel with a training set which is generated by randomly drawing, with replacement, N examples(or data) from the original training dataset, where N is the size of the original training set. The training set for each of the base classifiers is independent of each other. Many of the original data may be repeated in the resulting training set while others may be left out.

Bagging reduces overfitting (variance) by averaging or voting, however, this leads to an increase in bias, which is compensated by the reduction in variance though.



*Bagging classifier*

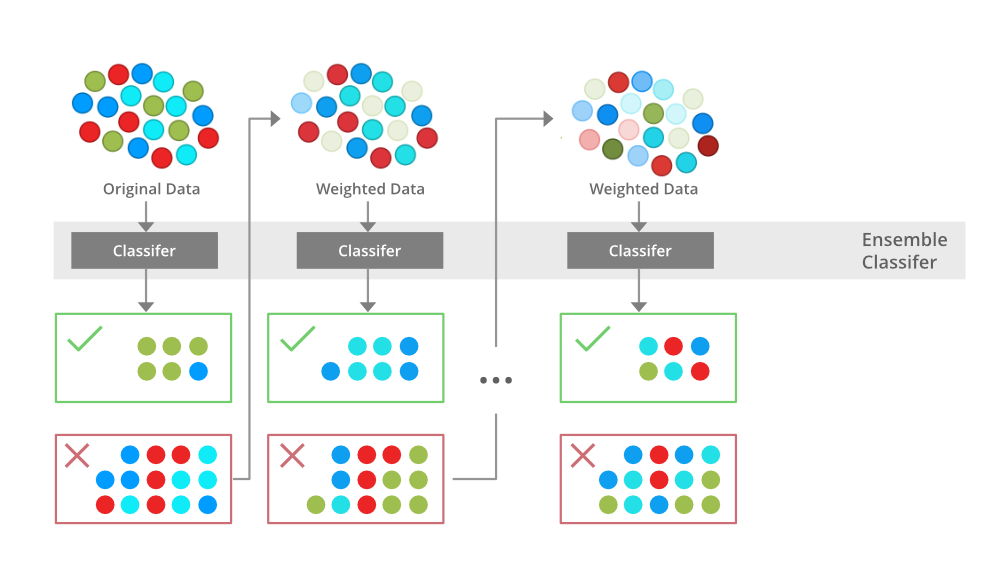
**Random Forest:**

Every decision tree has high variance, but when we combine all of them together in parallel then the resultant variance is low as each decision tree gets perfectly trained on that particular sample data and hence the output doesn’t depend on one decision tree but multiple decision trees. In the case of a classification problem, the final output is taken by using the majority voting classifier. In the case of a regression problem, the final output is the mean of all the outputs. This part is Aggregation.

The basic idea behind this is to combine multiple decision trees in determining the final output rather than relying on individual decision trees.  
Random Forest has multiple decision trees as base learning models. We randomly perform row sampling and feature sampling from the dataset forming sample datasets for every model. This part is called Bootstrap.

**Boosting:**

Boosting is an ensemble modelling, technique that attempts to build a strong classifier from the number of weak classifiers. It is done by building a model by using weak models in series. Firstly, a model is built from the training data. Then the second model is built which tries to correct the errors present in the first model. This procedure is continued and models are added until either the complete training data set is predicted correctly or the maximum number of models are added.



*Boosting*

**Gradient Boosting**

Gradient Boosting is a popular boosting algorithm. In gradient boosting, each predictor corrects its predecessor’s error. In contrast to Adaboost, the weights of the training instances are not tweaked, instead, each predictor is trained using the residual errors of predecessor as labels.

There is a technique called the Gradient Boosted Trees whose base learner is CART (Classification and Regression Trees).

**XGBoost**

XGBoost is an implementation of Gradient Boosted decision trees. XGBoost models majorly dominate in many Kaggle Competitions.

In this algorithm, decision trees are created in sequential form. Weights play an important role in XGBoost. Weights are assigned to all the independent variables which are then fed into the decision tree which predicts results. The weight of variables predicted wrong by the tree is increased and these variables are then fed to the second decision tree. These individual classifiers/predictors then ensemble to give a strong and more precise model. It can work on regression, classification, ranking, and user-defined prediction problems.

**Optimization and Improvement**

**System Optimization:**

1. **Regularization**: Since the ensembling of decisions, trees can sometimes lead to very complex. XGBoost uses both Lasso and Ridge Regression regularization to penalize the highly complex model.
2. **Parallelization and Cache block**: In, XGboost, we cannot train multiple trees parallel, but it can generate the different nodes of tree parallel. For that, data needs to be sorted in order. In order to reduce the cost of sorting, it stores the data in blocks. It stored the data in the compressed column format, with each column sorted by the corresponding feature value. This switch improves algorithmic performance by offsetting any parallelization overheads in computation.
3. **Tree Pruning:**  XGBoost uses max\_depth parameter as specified the stopping criteria for the splitting of the branch, and starts pruning trees backward. This depth-first approach improves computational performance significantly.
4. **Cache-Awareness and Out-of-score computation**: This algorithm has been designed to make use of hardware resources efficiently. This is accomplished by cache awareness by allocating internal buffers in each thread to store gradient statistics. Further enhancements such as ‘out-of-core computing optimize available disk space while handling big data-frames that do not fit into memory. In out-of-core computation, Xgboost tries to minimize the dataset by compressing it.
5. **Sparsity Awareness**: XGBoost can handle sparse data that may be generated from preprocessing steps or missing values. It uses a special split finding algorithm that is incorporated into it that can handle different types of sparsity patterns.
6. **Weighted Quantile Sketch:** XGBoost has in-built the distributed weighted quantile sketch algorithm that makes it easier to effectively find the optimal split points among weighted datasets.
7. **Cross-validation**: XGboost implementation comes with a built-in cross-validation method. This helps the algorithm prevents overfitting when the dataset is not that big,

**Advantages Or Disadvantages:**

**Advantages of XGBoost:**

1. Performance: XGBoost has a strong track record of producing high-quality results in various machine learning tasks, especially in Kaggle competitions, where it has been a popular choice for winning solutions.
2. Scalability: XGBoost is designed for efficient and scalable training of machine learning models, making it suitable for large datasets.
3. Customizability: XGBoost has a wide range of hyperparameters that can be adjusted to optimize performance, making it highly customizable.
4. Handling of Missing Values: XGBoost has built-in support for handling missing values, making it easy to work with real-world data that often has missing values.
5. Interpretability: Unlike some machine learning algorithms that can be difficult to interpret, XGBoost provides feature importances, allowing for a better understanding of which variables are most important in making predictions.

**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.

**Difference Between Random Forest and XGBoost**

**Summary Table**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Random Forest** | **XGBoost** |
| **Type** | Bagging | Boosting |
| **Tree Construction** | Parallel (independent trees) | Sequential (error-correcting trees) |
| **Speed** | Faster training, slower prediction | Slower training, faster prediction |
| **Regularization** | No explicit regularization | L1 and L2 regularization |
| **Hyperparameter Tuning** | Simple | Complex, requires tuning |
| **Performance** | Good | Often better |
| **Feature Importance** | Basic impurity-based importance | Detailed metrics (gain, coverage) |
| **Handling Missing Data** | Requires preprocessing | Handles natively |

**Conclusion**: Both Random Forest and XGBoost are powerful algorithms, and the choice depends on the specific use case. Random Forest is simpler and robust, making it suitable for quick implementations and smaller datasets. XGBoost, with its advanced techniques and optimization, is ideal for high-performance applications and complex datasets where maximum accuracy is needed.