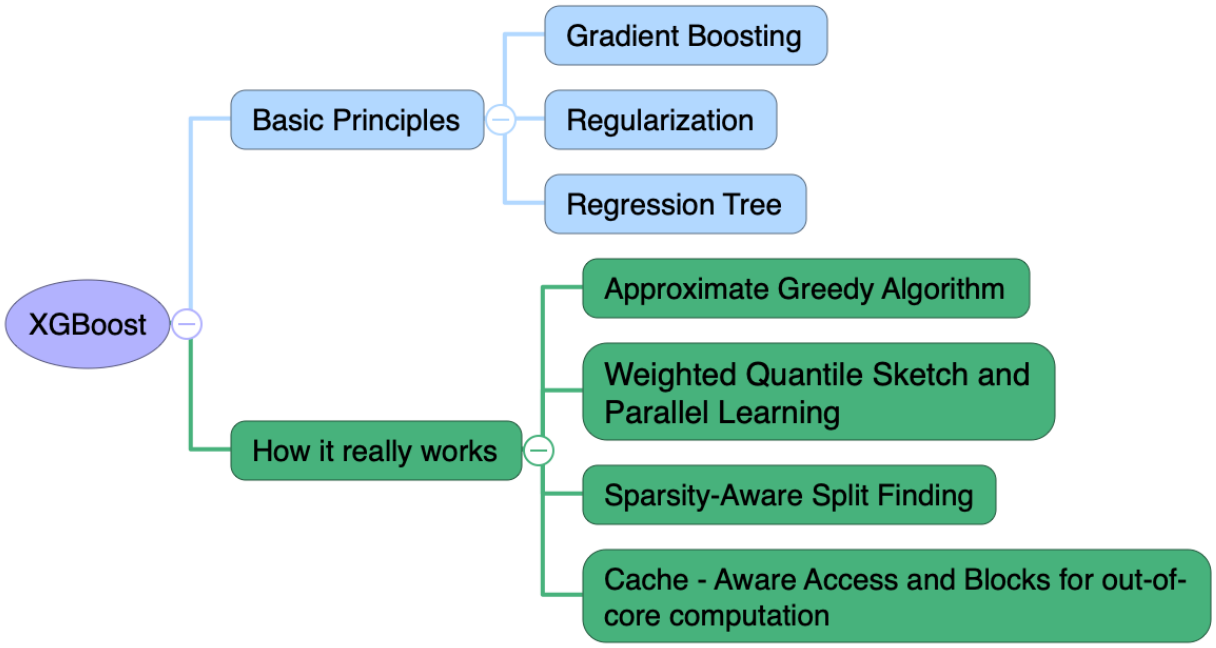
eXtreme Gradient Boosting



XGBoost stands for “Extreme Gradient Boosting”.

XGBoost is an **optimized distributed gradient boosting** library designed to be **highly efficient, flexible and portable.**

It **implements Machine Learning algorithms under the Gradient Boosting framework.**

It **provides a parallel tree boosting** to **solve many data science problems in a fast and accurate way.**

XGBoost is an **extension** to **gradient boosted decision trees (GBM)** and **specially designed** to **improve speed and performance**.

XGBoost Features

Regularization:

This is **considered** to be as a **dominant factor of the algorithm**. **Regularization** is a **technique** that is used to **get rid of overfitting of the model.**

Cross-Validation:

We use cross-validation by importing the function from sklearn but XGboost is enabled with inbuilt CV

Missing Value:

It is **designed in such a way that it can handle missing values**. It **finds out the trends in the missing values and apprehends them.**

Flexibility:

It gives the support to objective functions. They are the function used to evaluate the performance of the model and also it can handle the user-defined validation metrics.

Gaining speed

XGBoost was specifically designed for speed. Speed gains allow machine learning models to build more quickly which is especially important when dealing with millions, billions, or trillions of rows of data.

This is not uncommon in the world of big data, where each day, industry and science accumulate more data than ever before. The following new design features give XGBoost a big edge in speed over comparable ensemble algorithms:

Approximate split-finding algorithm

Sparsity aware split-finding

Parallel computing

Cache-aware access

Block compression and sharding

Approximate split-finding algorithm

Decision trees need **optimal splits** to **produce optimal results**.

A **greedy algorithm** selects the **best split at each step** and does not backtrack to look at previous branches. **bold text**

Note that **decision tree splitting** is **usually performed in a greedy manner**.

XGBoost presents an exact greedy algorithm in **addition to a new approximate split-finding algorithm.**

The **split-finding algorithm uses quantiles**, **percentages that split data, to propose candidate splits.**

In a global proposal, the **same quantiles** are used throughout the **entire training**, and in a **local proposal, new quantiles are provided for each round of splitting.**

Sparsity-aware splitfinding

Sparse matrices are designed to only store data points with non-zero and non-null values. This saves valuable space. A sparsityaware split indicates that when looking for splits, XGBoost is faster because its matrices are sparse

Parallel computing

Boosting is not ideal for parallel computing since **each tree depends on the results of the previous tree.**

There are opportunities, however, where parallelization may take place. Parallel computing occurs when multiple computational units are working together on the same problem at the same time.

XGBoost sorts and compresses the data into blocks. These blocks may be distributed to multiple machines, or to external memory (out of core).

Sorting the data is faster with blocks. The split-finding algorithm takes advantage of blocks and the search for quantiles is faster due to blocks.

In each of these cases, XGBoost provides parallel computing to expedite the model-building process.

Cache-aware access

The **data** on your computer is separated into **cache and main memory**.

The **cache**, what you use most often, is reserved for high-speed memory.

The data that **you use less often is held back for lower-speed memory.**

When it comes to gradient statistics, XGBoost uses cache-aware prefetching.

XGBoost allocates an internal buffer, fetches the gradient statistics, and performs accumulation with mini batches.

According to XGBoost:

A Scalable Tree Boosting System, prefetching lengthens read/write dependency and reduces runtimes by approximately 50% for datasets with a large number of rows.

Block compression and sharding

XGBoost delivers additional speed gains through block compression and block sharding.

Block compression helps with computationally expensive disk reading by compressing columns.

Block sharding decreases read times by sharding the data into **multiple disks that alternate when reading the data**

Step by Step implementation of XGBoost Algorithm

Trees in **other ensemble algorithm are created in the conventional manner i.e. either using Gini Impurity or Entropy**. But XGBoost introduces a n**ew metric called similarity score for node selection and splitting.**

Step:1

Create a **single leaf tree**.

Step:2

For the **first tree**, compute the **average of target variable as prediction** and **calculate the residuals using the desired loss function**.

For **subsequent trees the residuals come from prediction made by previous tree**.

Step:3

Calculate the s**imilarity score using the following formula**:

$$Similarity\ Score = \frac{Gradient^2}{Hessian + \lambda}$$

where, Hessian is equal to number of residuals; Gradient2 = squared sum of residuals; λ is a regularization hyperparameter.

Step:4

Using **similarity score we select the appropriate node**.**Higher the similarity score more the homogeneity.**

Step:5

Using **similarity score we calculate Information gain**.

Information gain gives the difference between old similarity and new similarity and thus tells how much homogeneity is achieved by splitting the node at a given point.

It is calculated using the following formula:

Information Gain = Left Similarity + Right Similarity – Similarity for Root

Step:6

Create the tree of desired length using the above method. **Pruning and regularization** would be done by **playing with the regularization hyperparameter.**

Step:7

Predict the **residual values using the Decision Tree you constructed**.

Step:8

The new set of residuals is calculated using the following formula:

$$New\ Residuals = Old\ Residulas + \rho \sum Predicted\ Residuals$$

Step:9

Go back to **step 1 and repeat the process for all the trees**

Analyzing XGBoost parameters

Learning objective

The **learning objective of a machine learning model determines how well the model fits the data.**

In the case of XGBoost, the learning objective consists of two parts:

the loss function and the regularization term.

$$obj(\theta) = l(\theta) + \Omega(\theta)$$

$l(\theta)$

is the loss function, which is the Mean Squared Error (MSE) for regression or the log loss for classification

$\Omega(\theta)$

is the regularization function, a penalty term to prevent over-fitting. Including a regularization term as part of the objective function distinguishes XGBoost from most tree ensembles.

Advantages and Disadvantages of XGBoost?

Advantages:

1.XGB consists of a number of hyper-parameters that can be tuned — a primary advantage over gradient boosting machines.

2.XGBoost has an in-built capability to handle missing values.

3.It provides various intuitive features, such as parallelisation, distributed computing, cache optimisation, and more.

Disadvantages:

1.Like any other boosting method, XGB is sensitive to outliers.

2.Unlike LightGBM, in XGB, one has to manually create dummy variable/ label encoding for categorical features before feeding them into the models.