**XGBoosting**

**What is XGBoost ? Why is it so good ?**

In the realm of data science, machine learning algorithms, and model building, the ultimate goal is to build the strongest predictive model while accounting for computational efficiency as well. This is where XGBoosting comes into play. XGBoost (eXtreme Gradient Boosting) is a direct application of Gradient Boosting for decision trees. There are a myriad of resources that dive into the mathematical backing and systematic functions of XGBoost.

XGBoost (**Ex**treme **G**radient **Boost**ing) is an optimized distributed gradient boosting library. Yes, it uses gradient boosting (GBM) framework at core. Yet, does better than GBM framework alone. XGBoost was created by [Tianqi Chen](http://homes.cs.washington.edu/~tqchen/), PhD Student, University of Washington. It is used for supervised ML problems.  Let's look at what makes it so good:

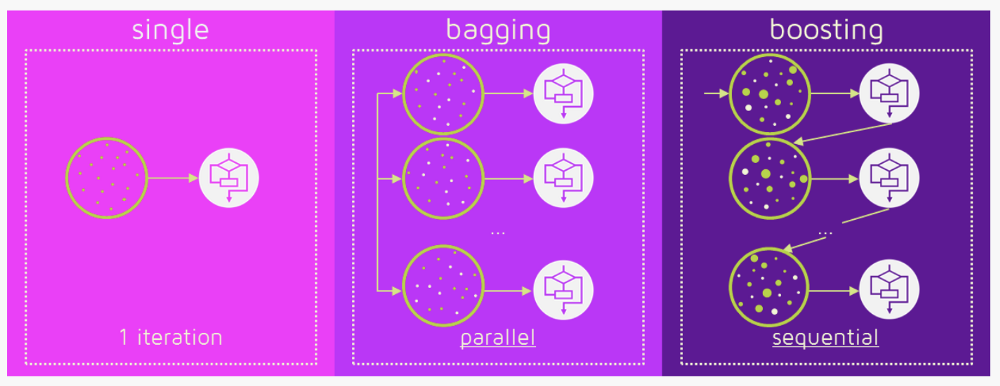
1. **Parallel Computing:** It is enabled with parallel processing (using OpenMP); i.e., when you run xgboost, by default, it would use all the cores of your laptop/machine.
2. **Regularization:** I believe this is the biggest advantage of xgboost. GBM has no provision for regularization. Regularization is a technique used to avoid overfitting in linear and tree-based models.
3. **Enabled Cross Validation:** In R, we usually use external packages such as caret and mlr to obtain CV results. But, xgboost is enabled with internal CV function (we'll see below).
4. **Missing Values:** XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.
5. **Flexibility:** In addition to regression, classification, and ranking problems, it supports user-defined objective functions also. An objective function is used to measure the performance of the model given a certain set of parameters. Furthermore, it supports user defined evaluation metrics as well.
6. **Availability:** Currently, it is available for programming languages such as R, Python, Java, Julia, and Scala.
7. **Save and Reload:** XGBoost gives us a feature to save our data matrix and model and reload it later. Suppose, we have a large data set, we can simply save the model and use it in future instead of wasting time redoing the computation.
8. **Tree Pruning:** Unlike GBM, where tree pruning stops once a negative loss is encountered, XGBoost grows the tree upto max\_depth and then prune backward until the improvement in loss function is below a threshold.

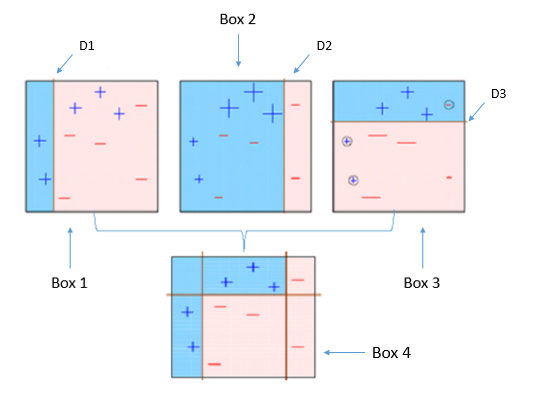
**How does XGBoost work ?**

XGBoost belongs to a family of boosting algorithms that convert weak learners into strong learners. A weak learner is one which is slightly better than random guessing. Let's understand **boosting first** (in general).

**BOOSTING** : The term ‘Boosting’ refers to a family of algorithms which converts weak learner to strong learners.

Boosting is a sequential process; i.e., trees are grown using the information from a previously grown tree one after the other. This process slowly learns from data and tries to improve its prediction in subsequent iterations. Let's look at a classic classification example:

****

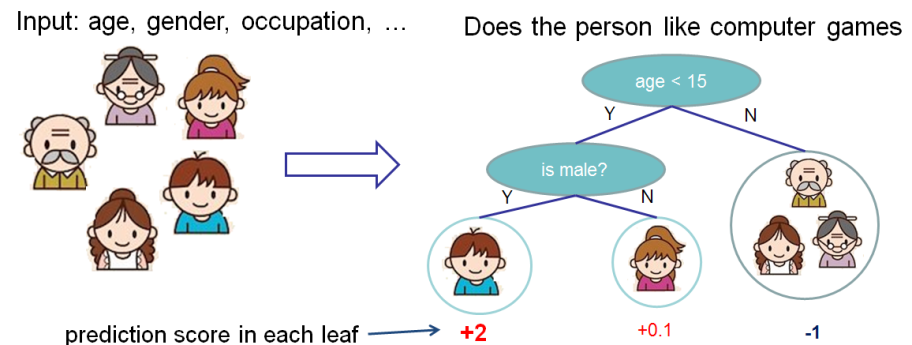
[](http://blog.hackerearth.com/wp-content/uploads/2016/12/bigd.png)

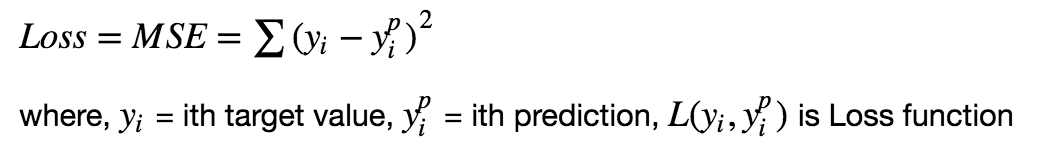
Four classifiers (in 4 boxes), shown above, are trying hard to classify + and - classes as homogeneously as possible. Let's understand this picture well.

1. **Box 1:** The first classifier creates a vertical line (split) at D1. It says anything to the left of D1 is + and anything to the right of D1 is -. However, this classifier misclassifies three + points.
2. **Box 2:** The next classifier says don't worry I will correct your mistakes. Therefore, it gives more weight to the three + misclassified points (see bigger size of +) and creates a vertical line at D2. Again it says, anything to right of D2 is - and left is +.  Still, it makes mistakes by incorrectly classifying three - points.
3. **Box 3:** The next classifier continues to bestow support. Again, it gives more weight to the three - misclassified points and creates a horizontal line at D3. Still, this classifier fails to classify the points (in circle) correctly.
4. Remember that each of these classifiers has a misclassification error associated with them.
5. Boxes 1,2, and 3 are weak classifiers. These classifiers will now be used to create a strong classifier Box 4.
6. **Box 4:** It is a weighted combination of the weak classifiers. As you can see, it does good job at classifying all the points correctly.

That's the basic idea behind boosting algorithms.

**Gradient boosting** : It is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees

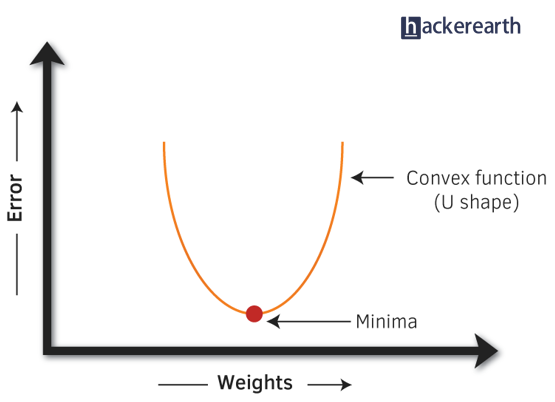




**Gradient Descent:**

**It’s an algo which iterate over weights(slope and intercept) in a optimal way to find minimum error.**

* It is a method which comprises a vector of weights (or coefficients) where we calculate their partial derivative with respective to zero. The motive behind calculating their partial derivative is to find the local minima of the loss function ([RSS](https://en.wikipedia.org/wiki/Residual_sum_of_squares)), which is convex in nature. In simple words, **gradient descent tries to optimize the loss function by tuning different values of coefficients to minimize the error.**

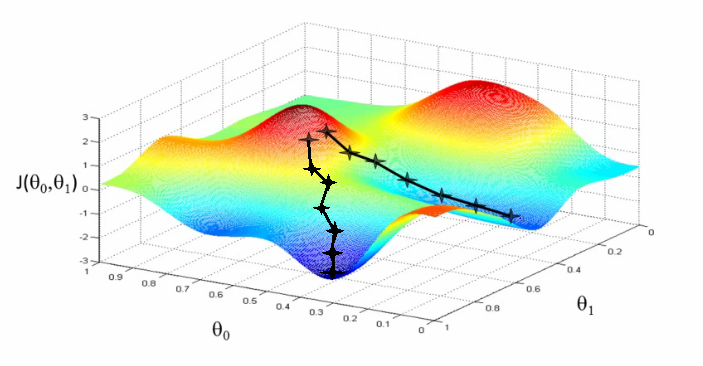
[](http://blog.hackerearth.com/wp-content/uploads/2016/12/graph.png)

https://www.youtube.com/watch?v=Trk-eZIwrSw

<https://www.youtube.com/watch?v=MlcRklxXxU8>

j(m,b) = loss/cost function = error square = summation of (actual y - predicted y) square

Xi+1 = Xi – alfa d/dx (j(m,b))  
where j(m,b) = x­2

****

So, the intuition behind gradient boosting algorithm is to repetitively leverage the patterns in residuals and strengthen a model with weak predictions and make it better. Once we reach a stage that residuals do not have any pattern that could be modeled, we can stop modeling residuals (otherwise it might lead to overfitting).

As we know, **XGBoost can used to solve both regression and classification problems**. It is enabled with separate methods to solve respective problems.

**Classification Problems:** To solve such problems, it uses booster = gbtree parameter; i.e., a tree is grown one after other and attempts to reduce misclassification rate in subsequent iterations. In this, the next tree is built by giving a higher weight to misclassified points by the previous tree.

**Regression Problems:** To solve such problems, we have two methods: booster = gbtree and booster = gblinear. You already know gbtree. In gblinear, it builds generalized linear model and optimizes it using regularization (L1,L2) and gradient descent. In this, the subsequent models are built on residuals (actual - predicted) generated by previous iterations.

Hopefully, up till now, you have developed a basic intuition around how boosting and xgboost works. Let's proceed to understand its parameters. After all, using xgboost without parameter tuning is like driving a car without changing its gears; you can never up your speed.

**Note:** In R, xgboost package uses a matrix of input data instead of a data frame.

**Understanding XGBoost Tuning Parameters**

Every parameter has a significant role to play in the model's performance. Before hypertuning, let's first understand about these parameters and their importance. In this article, I've only explained the most frequently used and tunable parameters. To look at all the parameters, you can refer to its [official documentation](http://xgboost.readthedocs.io/en/latest/parameter.html).

XGBoost parameters can be divided into three categories (as suggested by its authors):

* **General Parameters:** Controls the booster type in the model which eventually drives overall functioning
* **Booster Parameters:** Controls the performance of the selected booster
* **Learning Task Parameters:** Sets and evaluates the learning process of the booster from the given data

**1. General Parameters**

1. **Booster[default=gbtree]**
   * Sets the booster type (gbtree, gblinear or [dart](http://xgboost.readthedocs.io/en/latest/tutorials/dart.html)) to use. For classification problems, you can use gbtree, dart. For regression, you can use any.
2. **nthread[default=maximum cores available]**
   * Activates parallel computation. Generally, people don't change it as using maximum cores leads to the fastest computation.
3. **silent[default=0]**
   * If you set it to 1, your R console will get flooded with running messages. Better not to change it.

**2. Booster Parameters**

As mentioned above, parameters for tree and linear boosters are different. Let's understand each one of them:

**Parameters for Tree Booster**

1. **nrounds[default=100]**
   * **It controls the maximum number of iterations. For classification, it is similar to the number of trees to grow.**
   * Should be tuned using CV
2. **eta[default=0.3][range: (0,1)]**
   * **It controls the learning rate,** i.e., the rate at which our model learns patterns in data. After every round, it shrinks the feature weights to reach the best optimum.
   * Lower eta leads to slower computation. It must be supported by increase in nrounds.
   * Typically, it lies between 0.01 - 0.3
3. **gamma[default=0][range: (0,Inf)]**
   * **It controls regularization (or prevents overfitting).** The optimal value of gamma depends on the data set and other parameter values.
   * Higher the value, higher the regularization. Regularization means penalizing large coefficients which don't improve the model's performance. default = 0 means no regularization.
   * *Tune trick:* Start with 0 and check CV error rate. If you see train error >>> test error, bring gamma into action. Higher the gamma, lower the difference in train and test CV. If you have no clue what value to use, use gamma=5 and see the performance. Remember that gamma brings improvement when you want to use shallow (low max\_depth) trees.
4. **max\_depth[default=6][range: (0,Inf)]**
   * **It controls the depth of the tree.**
   * **Larger the depth, more complex the model; higher chances of overfitting.** There is no standard value for max\_depth. Larger data sets require deep trees to learn the rules from data.
   * Should be tuned using CV
5. **min\_child\_weight[default=1][range:(0,Inf)]**
   * In regression, it refers to the minimum number of instances required in a child node. In classification, if the leaf node has a minimum sum of instance weight (calculated by second order partial derivative) lower than min\_child\_weight, the tree splitting stops.
   * In simple words, it blocks the potential feature interactions to prevent overfitting. Should be tuned using CV.
6. **subsample[default=1][range: (0,1)]**
   * **It controls the number of samples (observations) supplied to a tree.**
   * Typically, its values lie between (0.5-0.8)
7. **colsample\_bytree[default=1][range: (0,1)]**
   * **It control the number of features (variables) supplied to a tree**
   * Typically, its values lie between (0.5,0.9)
8. **lambda[default=0]**
   * It controls L2 regularization (equivalent to Ridge regression) on weights. It is used to avoid overfitting.
9. **alpha[default=1]**
   * It controls L1 regularization (equivalent to Lasso regression) on weights. In addition to shrinkage, enabling alpha also results in feature selection. Hence, it's more useful on high dimensional data sets.

**Parameters for Linear Booster**

Using linear booster has relatively lesser parameters to tune, hence it computes much faster than gbtree booster.

1. **nrounds[default=100]**
   * It controls the maximum number of iterations (steps) required for gradient descent to converge.
   * Should be tuned using CV
2. **lambda[default=0]**
   * It enables Ridge Regression. Same as above
3. **alpha[default=1]**
   * It enables Lasso Regression. Same as above

**3. Learning Task Parameters**

These parameters specify methods for the loss function and model evaluation. In addition to the parameters listed below, you are free to use a customized objective / evaluation function.

1. **Objective[default=reg:linear]**
   * reg:linear - for linear regression
   * binary:logistic - logistic regression for binary classification. It returns class probabilities
   * multi:softmax - multiclassification using softmax objective. It returns predicted class labels. It requires setting num\_class parameter denoting number of unique prediction classes.
   * multi:softprob - multiclassification using softmax objective. It returns predicted class probabilities.
2. **eval\_metric [no default, depends on objective selected]**
   * These metrics are used to evaluate a model's accuracy on validation data. For regression, default metric is RMSE. For classification, default metric is error.
   * Available error functions are as follows:
     + mae - Mean Absolute Error (used in regression)
     + Logloss - Negative loglikelihood (used in classification)
     + AUC - Area under curve (used in classification)
     + RMSE - Root mean square error (used in regression)
     + error - Binary classification error rate [#wrong cases/#all cases]
     + mlogloss - multiclass logloss (used in classification)

XGBoost is an advanced gradient boosted tree algorithm. It has support for parallel processing, regularization, early stopping which makes it a very fast, scalable and accurate algorithm.

Parameters:

* **Maximum number of trees:** XGBoost has an early stop mechanism so the exact number of trees will be optimized. High number of actual trees will increase the training and prediction time. Typical values: 100 - 10000
* **Early stopping:** Use XGBoost’s built-in early stop mechanism so the exact number of trees will be optimized. The cross-validation scheme defined in the **Train & validation** tab will be used.
* **Early stopping rounds:** The optimizer stops if the loss never decreases for this consecutive number of iterations. Typical values: 1 - 100
* **Maximum depth of tree:** Maximum depth of each tree. High values can increase the quality of the prediction, but can lead to overfitting. Typical values: 3 - 10. You can try multiple values by providing a comma-separated list. This increases the training time.
* **Learning rate:** Lower values slow down convergence and can make the model more robust. Typical values: 0.01 - 0.3. You can try multiple values by providing a comma-separated list. This increases the training time.
* **L2 regularization:** L2 regularization reduces the size of the coefficient for each feature. You can try multiple values by providing a comma-separated list. This increases the training time.
* **L1 regularization:** In addition to reduce overfitting, may improve scoring speed for very high dimensional datasets. You can try multiple values by providing a comma-separated list. This increases the training time.
* **Gamma:** Minimum loss reduction to split a leaf. You can try multiple values by providing a comma-separated list. This increases the training time.
* **Minimum child weight:** Minimum sum of weights(hessian) in a node. High values can prevent overfitting by learning highly specific cases. Smaller values allow leaf nodes to match a small set of rows, which can be relevant for highly imbalanced sets. You can try multiple values by providing a comma-separated list. This increases the training time.
* **Subsample:** Subsample ratio for the data to be used in each tree. Low values can prevent overfitting but can make specific cases harder to learn. Typical values: 0.5 - 1. You can try multiple values by providing a comma-separated list. This increases the training time.
* **Colsample by tree:** Fraction of the features to be used in each tree. Typical values: 0.5-1. You can try multiple values by providing a comma-separated list. This increases the training time.
* **Replace missing values:**
* **Parallelism:** Number of cores used for parallel training. Using more cores leads to faster training but at the expense of more memory consumption, especially for large training datasets. (-1 means “all cores”)

ThankYou

DMatrices : Instead of numpy arrays or pandas dataFrame, XGBoost uses DMatrices. A DMatrix can contain both the features and the target. If you already have loaded you data into numpy arrays X and y

mean absolute error (MAE)

num\_boost\_round and corresponds to the number of boosting rounds or trees to build.

N is defined by the variable early\_stopping\_round

* seed: random seed. It's important to set a seed here, to ensure we are using the same folds for each step so we can properly compare the scores with different parameters.
* nfold: the number of folds to use for cross-validation
* metrics: the metrics to use to evaluate our model, here we use MAE.

**­­­---**

**2. Gradient Boosting: Similar to all boosting methods, Gradient Boosting looks to consecutively reduce error with each consecutive model, until one final model is produced. Given a set of data observations, Gradient Boosting fits a simple, weak learner to predict outcomes. Then, from this weak model, a loss function is plotted. There are various loss functions we can use in machine learning, but the ultimate goal of each loss function is to reduce error. Next, the two plots?—?the original data plot and the loss function?—?are combined to make a stronger predictor. The sum of our predictors gets stronger and stronger after each step. This process is repeated until a final predictor is built. Take the diagram below. The “Ground Truth” plots a set of data, with a line running through each of the points.**

**2’. This is the cycle of one weak learner in Gradient Boosting. By combining weak learner after weak learner, our final model is able to account for a lot of the error from the original model and reduces this error over time.**

**Gradient Boosting gets its name from Gradient Descent. Given the predetermined loss function, Gradient Descent is utilized to find the parameters which minimize this loss function. Initially, gradient descent uses some parameters to looks at each point along the loss function, and find the negative derivative of that point. As gradient descent continues along the loss function, it continuously tunes the parameters until the minimum point is found. The goal is to find the optimal parameters which have the biggest decrease on the loss function. This is how Gradient Boosting attempts to minimize error. By sequentially minimizing our loss function (meaning we are sequentially minimizing the amount of error with each weak learner), our model gets stronger and stronger until a final predictor is found.**