*MACHINE LEARNING MODELS*

# **XGBOOST**

## *Gradient Boosting*

### Definition

*Gradient boostin*g is one of the most powerful techniques for building predictive models. It is an automatic learning technique used for regression analysis and statistical classification problems, which produces a predictive model in the form of a set of weak predictive models, typically decision trees. It builds the model in a staggered manner as other boosting methods do, and generalizes them allowing arbitrary optimization of a differentiable loss function.

This algorithm is an efficient algorithm for converting relatively poor hypotheses into very good hypotheses. A **weak hypothesis** or weak learner is defined as one whose performance is at least slightly better than random chance. *Hypothesis boosting* was the idea of filtering observations, leaving those observations that the weak learner can handle and focusing on developing new weak learns to handle the remaining difficult observations. The idea is to use the weak learning method several times to get a succession of hypotheses, each one refocused on the examples that the previous ones found difficult and misclassified.

Gradient boosting involves three elements:

1. A **loss function** to be optimized.
2. A **weak learner** to make predictions.
3. An **additive model** to add weak learners to minimize the loss function.

#### 

#### Loss Function

A **loss function** or **cost function** is a function that maps an event or values of

one or more variables onto a real number intuitively representing some *"cost"* associated with the event. Anoptimization problem seeks to minimize a loss function.

The *loss function* used depends on the type of problem being solved.

It must be *differentiable* (a **differentiable function** of one real variable is a function whose derivative exists at each point in its domain), but many standard loss functions are supported and you can define your own. For example, regression may use a squared error and classification may use logarithmic loss.

A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

#### Weak Learner

Decision trees are used as the weak learner in gradient boosting. Specifically regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and “correct” the residuals in the predictions.

Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss. Initially, very short decision trees were used that only had a single split, called a decision stump. Larger trees can be used generally with 4-to-8 levels. It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes.

This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

#### Additive Model

Trees are added one at a time, and existing trees in the model are not changed. A gradient descent procedure is used to minimize the loss when adding trees. Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.

Instead of parameters, we have **weak learner sub-models** or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must **add a tree to the model** that reduces the loss (i.e. follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by reducing the residual loss.

Generally this approach is called **functional gradient descent** or **gradient descent with functions**. One way to produce a weighted combination of classifiers which optimizes the cost is by gradient descent in function space.

A fixed number of trees are added or training stops once loss reaches an acceptable level or no longer improves on an external validation dataset.

### Improvements to Basic Gradient Boosting

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting. In this this section we will look at 4 enhancements to basic gradient boosting:

1. *Tree Constraints*
2. *Shrinkage*
3. *Random sampling*
4. *Penalized Learning*

#### Tree Constraints

It is important that the weak learners have skill but remain weak. There are a number of ways that the trees can be constrained. A good general heuristic is that the more constrained tree creation is, the more trees you will need in the model, and the reverse, where less constrained individual trees, the fewer trees that will be required.

Below are some constraints that can be imposed on the construction of decision trees:

* **Number of trees**, generally adding more trees to the model can be very slow to overfit. The advice is to keep adding trees until no further improvement is observed.
* **Tree depth**, deeper trees are more complex trees and shorter trees are preferred. Generally, better results are seen with 4-8 levels.
* **Number of nodes or number of leaves**, like depth, this can constrain the size of the tree, but is not constrained to a symmetrical structure if other constraints are used.
* **Number of observations per split** imposes a minimum constraint on the amount of training data at a training node before a split can be considered
* **Minimum improvement to loss** is a constraint on the improvement of any split added to a tree.

#### Weighted Updates

The predictions of each tree are added together sequentially. The contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called a **shrinkage** or a **learning rate**. Each update is simply scaled by the value of the “learning rate parameter v”.

The effect is that learning is slowed down, in turn require more trees to be added to the model, in turn taking longer to train, providing a configuration trade-off between the number of trees and learning rate. Decreasing the value of v [the learning rate] increases the best value for M [the number of trees]. It is common to have small values in the range of 0.1 to 0.3, as well as values less than 0.1.

Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model.

#### Stochastic Gradient Boosting

A big insight into bagging ensembles and random forest was allowing trees to be greedily created from subsamples of the training dataset. This same benefit can be used to reduce the correlation between the trees in the sequence in gradient boosting models. This variation of boosting is called **stochastic gradient boosting**.

At each iteration a subsample of the training data is drawn at random (without replacement) from the full training dataset. The randomly selected subsample is then used, instead of the full sample, to fit the base learner.

A few variants of stochastic boosting that can be used:

* Subsample rows before creating each tree.
* Subsample columns before creating each tree
* Subsample columns before considering each split.

Generally, aggressive sub-sampling such as selecting only 50% of the data has shown to be beneficial.

#### Penalized Gradient Boosting

Additional constraints can be imposed on the parameterized trees in addition to their structure. Classical decision trees like CART are not used as weak learners, instead a modified form called a regression tree is used that has numeric values in the leaf nodes (also called terminal nodes). The values in the leaves of the trees can be called weights in some literature.

As such, the leaf weight values of the trees can be regularized using popular regularization functions, such as:

* L1 regularization of weights.
* L2 regularization of weights.

(See functions [here](https://msdn.microsoft.com/en-us/magazine/dn904675.aspx))

The additional regularization term helps to smooth the final learnt weights to avoid over-fitting. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions.

XGBoost is an implementation of ***gradient boosted decision trees*** designed for speed and performance that is dominative competitive machine learning.

* It now has integrations with *scikit-learn* for Python users
* <https://machinelearningmastery.com/develop-first-xgboost-model-python-scikit-learn/> (How to Develop Your First XGBoost Model in Python with scikit-learn)
* <https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/> (A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning)
* <https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/> (A Gentle Introduction to XGBoost for Applied Machine Learning)

## **LightGBM**

* <https://lightgbm.readthedocs.io/en/latest/>
* LightGBM uses the leaf-wise (best first) tree growth algorithm, while many other popular tools use depth-wise tree growth. Compared with depth-wise growth, the leaf-wise algorithm can converge much faster. However, the leaf-wise growth may be overfitting if not used with the appropriate parameters.
* LightGBM is a great implementation that is similar to XGBoost but varies in a few specific ways, especially in how it creates the trees.
* It offers some different parameters but most of them are very similar to their XGBoost counterparts.
* If you use the same parameters, you almost always get a very close score. In most cases, the training will be 2-10 times faster thoug
* *Why don't more people use it then?* XGBoost has been around longer and is already installed on many machines. LightGBM is rather new and didn't have a Python wrapper at first. The current version is easier to install and use so no obstacles here. Many of the more advanced users on Kaggle and similar sites already use LightGBM and for each new competition, it gets more and more coverage. Still, the starter scripts are often based aroudn XGBoost as people just reuse their old code and adjust a few parameters. I'm sure this will increase once there are a few more tutorials and guides on how to use it (most of the non-ScikitLearn guides currently focus on XGBoost or neural networks).
* <https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/> -> Comparison between XGBoost and LightGBM

## **Multiple Layer Perception**

* A **multilayer perceptron** (MLP) is a class of feedforward artificial neural network. An MLP consists of at least three layers of nodes. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable.
* <https://machinelearningmastery.com/neural-networks-crash-course/>
* <https://www.springboard.com/blog/beginners-guide-neural-network-in-python-scikit-learn-0-18/>
* Multi-layer Perceptron is sensitive to feature scaling.
* <http://benalexkeen.com/feature-scaling-with-scikit-learn/> -> Feature Scaling with scikit-learn