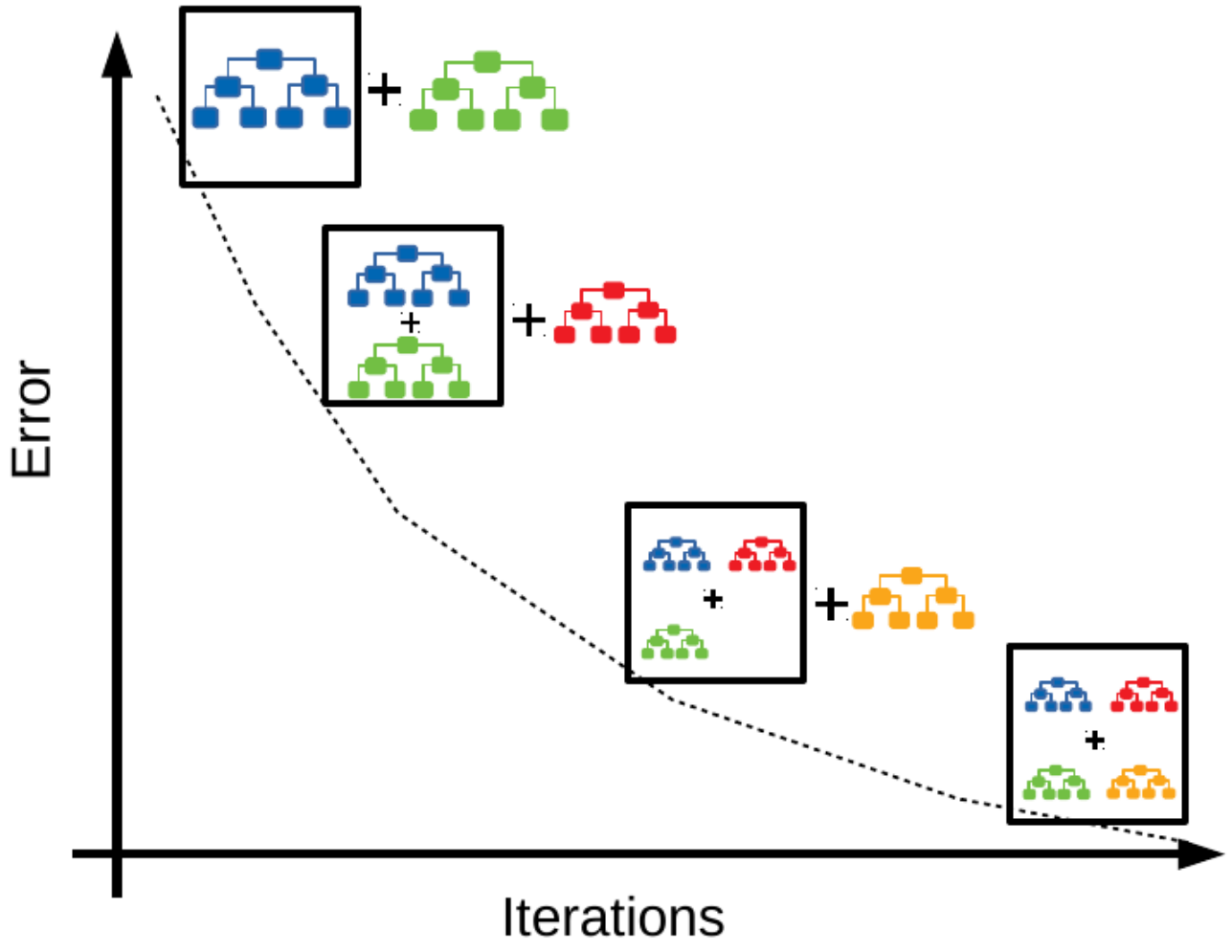
Gradient Tree Boosting

Gradient Boosting = Gradient Descent + Boosting



Types of Boosting Algorithms

AdaBoost (Adaptive Boosting)

Gradient Tree Boosting

XGBoost

History

Generalization of AdaBoost as Gradient Boosting

AdaBoost and related algorithms were recast in a statistical framework first by **Breiman calling them ARCing algorithms.**

Arcing is an acronym for **Adaptive Reweighting and Combining**.

Each step in an **arcing** algorithm consists of a **weighted minimization followed by a recomputation of [the classifiers] and [weighted input]**.

This **framework** was **further developed by Friedman** and **called Gradient Boosting Machines.** Later called just gradient boosting or gradient tree boosting.

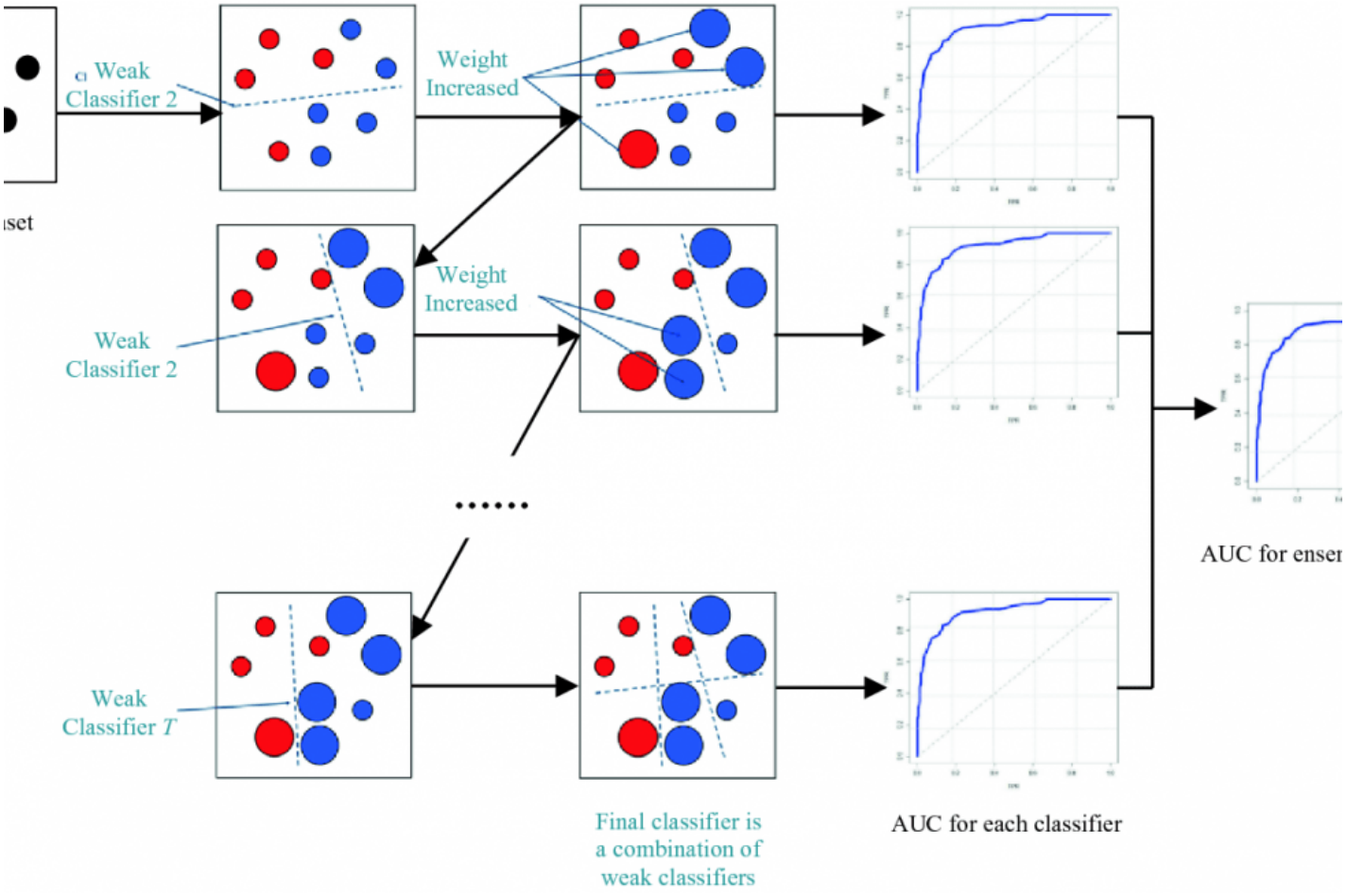
How Gradient Boosting Works

Gradient boosting involves **three elements:**

A **loss function** to be **optimized**.

A **weak learner** to make **predictions**.

An **additive model to add weak learners** to **minimize the loss function**.



Loss Function

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

It must be **differentiable**, 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, such as in the case of AdaBoost, 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 p**erform 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.**

The output for the new tree is then added to the output of the existing sequence of trees in an effort to correct or improve the final output of the model.

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.**

We will look at 4 enhancements to basic gradient boosting:

Tree Constraints

Shrinkage

Random sampling

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** Minimim 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.

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.

It is common to have small values in the range of 0.1 to 0.3, as well as values less than 0.1.

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.

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.

Hyperparameters

A simple GBM model contains two categories of hyperparameters:

boosting hyperparameters and

tree-specific hyperparameters.

The two main boosting hyperparameters include:

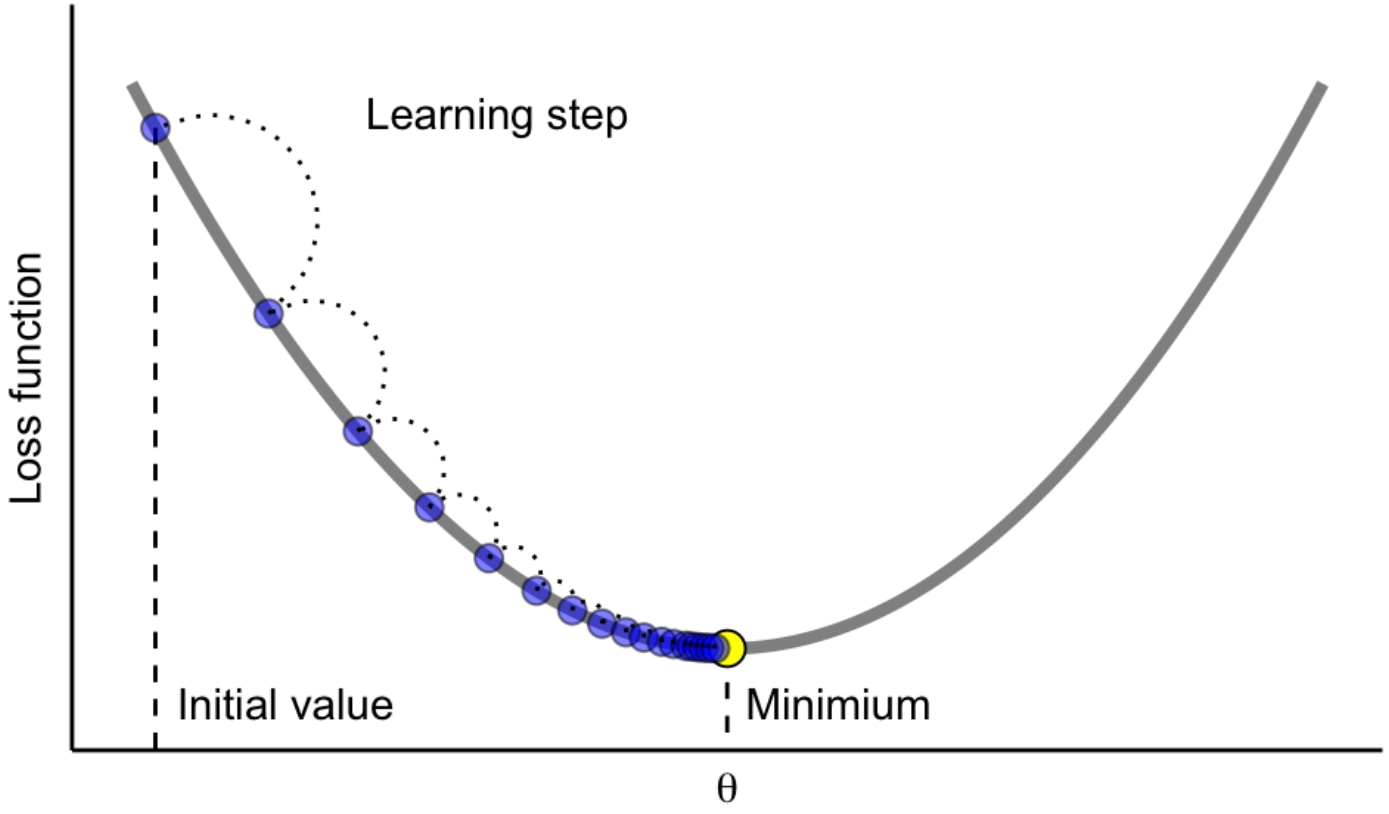
Number of trees:

The total number of trees in the sequence or ensemble.

The averaging of independently grown trees in bagging and random forests makes it very difficult to overfit with too many trees.

Learning rate:

Determines the contribution of each tree on the final outcome and controls how quickly the algorithm proceeds down the gradient descent (learns).



The two main tree hyperparameters in a simple GBM model include:

Tree depth:

Controls the depth of the individual trees. Typical values range from a depth of 3–8 but it is not uncommon to see a tree depth of 1.

Smaller depth trees such as decision stumps are computationally efficient (but require more trees)

Minimum number of observations in terminal nodes:

Also, controls the complexity of each tree. Since we tend to use shorter trees this rarely has a large impact on performance.

Typical values range from 5–15 where higher values help prevent a model from learning relationships which might be highly specific to the particular sample selected for a tree (overfitting) but smaller values can help with imbalanced target classes in classification problems.

General tuning strategy

Unlike random forests, GBMs can have high variability in accuracy dependent on their hyperparameter settings.

So tuning can require much more strategy than a random forest model. Often, a good approach is to:

1.**Choose a relatively high learning rate**. Generally the default value of 0.1 works but somewhere between 0.05–0.2 should work across a wide range of problems.

2.**Determine the optimum number of trees** for this learning rate.

3.**Fix tree hyperparameters and tune learning rate** and **assess speed vs. performance.**

4.**Tune tree-specific parameters** for **decided learning rate**.

5.**Once tree-specific parameters have been found**, **lower the learning rate to assess for any improvements in accuracy.**

6.**Use final hyperparameter settings** and **increase CV procedures to get more robust estimates.** Often, **the above steps are performed with a simple validation procedure or 5-fold CV due to computational constraints.**

If you used k-fold CV throughout steps 1–5 then this step is not necessary.

Advantages of Gradient Boosting

Better accuracy:

Gradient Boosting Regression generally provides better accuracy.

Less pre-processing:

As we know that data pre processing is one of the vital steps in machine learning workflow, and if we do not do it properly then it affects our model accuracy.

However, Gradient Boosting Regression requires minimal data preprocessing, which helps us in implementing this model faster with lesser complexity.

Higher flexibility:

Gradient Boosting Regression provides can be used with many hyper-parameter and loss functions.

This makes the model highly flexible and it can be used to solve a wide variety of problems.

Missing data:

Missing data is one of the issue while training a model.

Gradient Boosting Regression handles the missing data on its own and does not require us to handle it explicitly.

In this algorithm the missing values are treated as **containing information**.

Thus during tree building, **splitting decisions for node are decided by minimizing the loss function** and **treating missing values as a separate category that can go either left or right.**
