## **Introduction to Boosting**

Boosting is one of the most powerful ideas introduced in the field of machine learning in the past few years. It was first introduced in 1997 by Freund and Schapire in the popular algorithm, AdaBoost. It was originally designed for classification problems. Since its inception, many new boosting algorithms have been developed those tackle regression problems also and have become famous as they are used in the top solutions of many Kaggle competitions. We will go through the concepts of the most popular boosting algorithms - AdaBoost, Gradient Boosting and XGBoost in this module.

|  |  |
| --- | --- |
|  | Here, we would prefer model 1, instead of model 2. The key idea of boosting is to create an ensemble which makes high errors only on the less frequent data points. |

Boosting leverages the fact that we can build a series of models specifically targeted at the data points which have been incorrectly predicted by the other models in the ensemble. If a series of models keep reducing the average error, we will have an ensemble having extremely high accuracy.

Boosting is a way of generating a strong model from a weak learning algorithm. Note that here SVM, regression, and other techniques are algorithms which are used to create models. Prof. Raghavan briefly mentions the function of the algorithm. It is to minimize the loss.

 At this point, it is important to understand the loss functions for regression and classification problems are different. Until now, we have defined the error function for a regression setting as the sum of squared difference between the actual and the predicted values while the misclassification rate as the error function for a classification problem. We will see in the upcoming segments how the loss function is a modification of these error functions.

A weak learning algorithm produces a model that does marginally better than a random guess. A random guess has a 50% chance of being right. Hence, any such model shall have, say 60-70% chance of being correct and the final objective is to create a strong model by making an ensemble of such weak models.

# Boosting

Boosting methods work in the same spirit as bagging methods: we build a family of models that are aggregated to obtain a strong learner that performs better. However, unlike bagging that mainly aims at reducing variance, boosting is a technique that consists in fitting sequentially multiple weak learners in a very adaptative way: each model in the sequence is fitted giving more importance to observations in the dataset that were badly handled by the previous models in the sequence. Intuitively, each new model **focus its efforts on the most difficult observations** to fit up to now, so that we obtain, at the end of the process, a strong learner with lower bias (even if we can notice that boosting can also have the effect of reducing variance). Boosting, like bagging, can be used for regression as well as for classification problems.

Being **mainly focused at reducing bias**, the base models that are often considered for boosting are models with low variance but high bias. For example, if we want to use trees as our base models, we will choose most of the time shallow decision trees with only a few depths. Another important reason that motivates the use of low variance but high bias models as weak learners for boosting is that these models are in general less computationally expensive to fit (few degrees of freedom when parametrized). Indeed, as computations to fit the different models **can’t be done in parallel** (unlike bagging), it could become too expensive to fit sequentially several complex models.

Once the weak learners have been chosen, we still need to define how they will be sequentially fitted (what information from previous models do we take into account when fitting current model?) and how they will be aggregated (how do we aggregate the current model to the previous ones?). We will discuss these questions in the two following subsections, describing more especially two important boosting algorithms: adaboost and gradient boosting.

In a nutshell, these two meta-algorithms differ on how they create and aggregate the weak learners during the sequential process. Adaptive boosting updates the weights attached to each of the training dataset observations whereas gradient boosting updates the value of these observations. This main difference comes from the way both methods try to solve the optimization problem of finding the best model that can be written as a weighted sum of weak learners.

# Which is the best, Bagging or Boosting?

There’s not an outright winner; it depends on the data, the simulation and the circumstances.  
Bagging and Boosting decrease the variance of your single estimate as they combine several estimates from different models. So the result may be a model with **higher stability**.

If the problem is that the single model gets a very low performance, Bagging will rarely get a **better bias**. However, Boosting could generate a combined model with lower errors as it optimizes the advantages and reduces pitfalls of the single model.

By contrast, if the difficulty of the single model is **over-fitting**, then Bagging is the best option. Boosting for its part doesn’t help to avoid over-fitting; in fact, this technique is faced with this problem itself. For this reason, Bagging is effective more often than Boosting.

# AdaBoost

AdaBoost algorithm boosts the probability of points that are incorrectly classified and suppresses the probability of points that are correctly classified. In other words, the distribution changes after every iteration.

Objective Function  =  minhED[L(h(xi),yi)] = minh∑ni=1PD(xi).L(h(xi),yi)

We see here that the with each new model, the distribution of the data changes. By distribution, we mean that the weight assigned to each data point changes for the calculation of objective function that needs to be minimized. Since, the above expression stands true for uniform distribution as PD(xi) = 1/n for a uniform distribution, it is nor expressed as  minh1/n∑ni=1L(h(xi),yi). In other words, objective function is the expected value of the loss which transforms to the above-mentioned objective function when the distribution is not uniform.

|  |  |
| --- | --- |
|  | there are ***essentially two steps involved in the AdaBoost algorithm***:   1. Modify the current distribution to create a new distribution to generate a new model 2. Calculation of the weights given to each of the models to get the final ensemble   At an iteration t, we have a distribution Dt of the training data T on which we fit a model Ht  and then use the results to create a new distribution Dt+1. The final model H we build is an ensemble of all the individual models Hi with weights αi.  Now we need to find the weight α associated with each of these model Hi |

**Probability Boosting**

In a binary classification into +/-1, at an iteration t, Ft predicts the values for certain data points as follows:

|  |  |  |
| --- | --- | --- |
| Data points | Predicted Value | Actual Value |
| x1 | -1 | +1 |
| x2 | +1 | +1 |
| x3 | -1 | -1 |
| x4 | +1 | -1 |

By looking at the table, probabilities of which of the points will be boosted?

**Ans:** x1, x4 as these points are incorrectly predicted, hence the probability for the next iteration will be increased.

**Probability Distribution**

Now that you have understood that the points which were incorrectly predicted have their probability boosted up, what type of function would best describe such behavior?

**Suggested Answer**

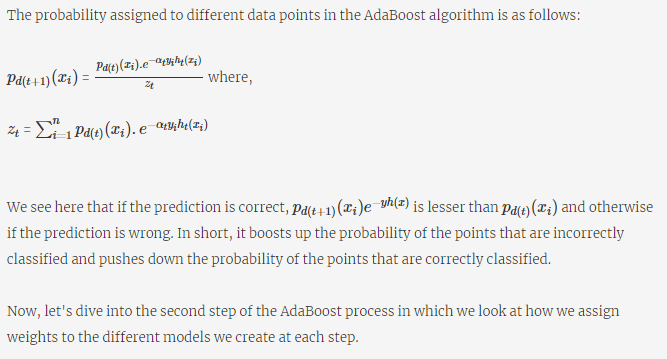
*We have 2 conditions:*

1. *The values given by the function should be positive as the probabilities cannot be negative*
2. *The function should have a higher value for negative input (incorrectly predicted) than a positive input (correctly predicted).*

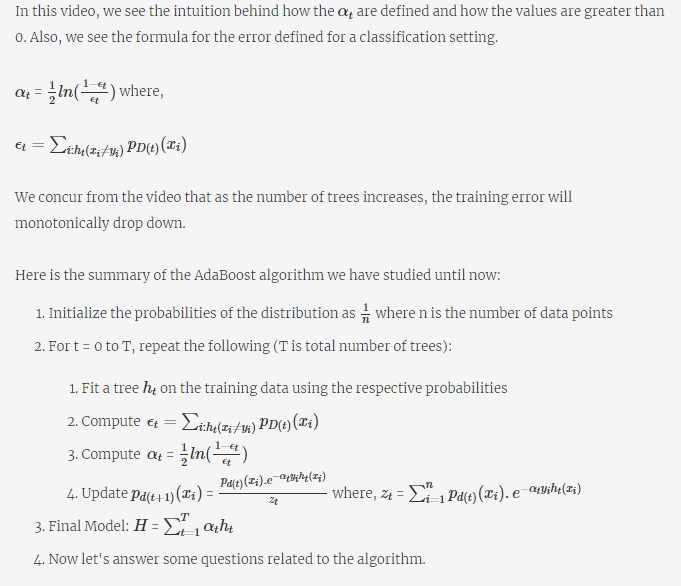
*Some of the functions that satisfy this function for an input x are:*

1. e−x
2. 2−x

|  |  |
| --- | --- |
| Here the sign of h(x) is same as sign of y. Which means the product of these would be > 0 i.e. y.h(x) > 0. We say that intuitively a model h does well on data point x if the product y.h(x) > 0.  ht which is the previous model a time t ahead a model is ht. If ht gave the correct prediction of xi then this yihtxi must be > 0 because it is giving the correct prediction. If this yihtxi > 0 then αt also > 0 where alpha α is a constant. These are the constants with which models are going to be combined with. Now the exponents of these quantity αtyihtxi is negative < 1 | This stands to a reason, if my model has already taken care of xi correctly then I need not to worry for that particular xi. So I bring that xi probability down and give it to the algorithm to produce the next model. Why because I don’t want to the model focus xi anymore as that data point it already being taken care of. On the other hand if ht behaves incorrectly on xi then the expression αtyihtxi is < 0, which means the exponent e of negative αtyihtxi is +ve. The expression αtyihtxi > 1 means the earlier probability is going to be the less than the next probability. This has the effect of jacking up or booting the probability of the data points which are misclassified and it has the effect of pushing the probability down the probability of the data points for correctly classified by the previous model. This is how we tweak the distribution in every iteration.  The Zt in denominator is a constant ensures that probabilities are less than < 0 and all these probabilities are add up to 1. Which means the probability of xi under new distribution is < than the probability of the xi under old distribution. |

****

|  |  |
| --- | --- |
| αt is the probability of of ht to make the mistakes. Each ht obviously not do correctly in all the data points. So we look at all the data points on which ht made a mistakes and you add the probabilities across those data points. | ϵt=∑i:ht(xi≠yi)pD(t)(xi) < 1/2  So i.e. epsilon ϵt is the sum over all the data points for which the predictions made by model ht was not the same as the ground truth. And each of these data points which was misclassified by ht had a probability under the distribution dt of iteration t.  We say our model is weak learner. This means it does marginally better job than random guess. To do the job better the error term ϵt should be < ½. For example we have a binary classification problem and if we need to do better than a random guess then the probability of error for my model should be less than < ½.  With this choice of αt as we go through the iterations the training error drops down monotonically. We can also show in lot of cases the generalization error also drops as we increase the number of model in the ensemble. |

****

***Why is the error ϵt always less than 1/2 in the above AdaBoost setting?***

**Suggested Answer :**

The models used in the Adaboost setting are weak learners but the prediction error they make is less than a random guess. A random guess error is 50% or 0.5. Hence, ϵt < 0.5

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Error value**  In a binary classification into +/-1, at an iteration t, the first model h0 predicts the values for certain data points as follows:   |  |  |  | | --- | --- | --- | | Data points | Predicted Value | Actual Value | | 1 | -1 | +1 | | 2 | +1 | +1 | | 3 | -1 | -1 | | 4 | +1 | -1 |   If we consider a hypothetical situation such that these data points constitute the complete dataset on which we build the model h0, is h0 be an acceptable starting model? | **No**  **Feedback :**  *We see that the model has misclassified 2 points and the misclassification error is 0.5. We cannot accept this model because the weak learners should have an error less than 0.5.* |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Probability Distribution**  The following table shows the actual value and the predicted value by an intermediate tree ht for 5 sample points :   |  |  |  | | --- | --- | --- | | Data points | Actual | Predicted | | 1 | +1 | -1 | | 2 | -1 | -1 | | 3 | +1 | +1 | | 4 | -1 | +1 | | 5 | +1 | +1 |   Probability distribution of which of the above points will be boosted for the next iteration? | **1,4**  **Feedback :**  It is misclassified, hence the probability will be boosted. |
| **αt​ value**  The following table shows the actual value and the predicted value by an intermediate tree ht for 5 sample points :   |  |  |  | | --- | --- | --- | | Data points | Actual | Predicted | | 1 | +1 | -1 | | 2 | -1 | -1 | | 3 | +1 | +1 | | 4 | -1 | +1 | | 5 | +1 | +1 | | If we consider a hypothetical situation such that these data points constitute the complete dataset on which we build the model ht, what weight αt will be attached to ht?  **0.2027**  **Feedback :**  *We see that the error*ϵt*= 2/5 = 0.4. Since 2 data points incorrectly predicted over 5*  *Setting this value into the formula, we get 1/2ln(0.6/0.4) = 0.2027* |

**Which method is used for creating an ensemble of models in Random Forest?**

Bagging

**Which of the following points are correct?**

**The weights assigned to each tree to create an ensemble are different in case of AdaBoost**

**Feedback :** The  are the weights assigned to each tree.

**The weights assigned to each tree to create an ensemble are same in case of Random forest**

**Feedback :** All trees have the same weights

**The AdaBoost algorithm uses weak learners to build models**

**Feedback :** As discussed in the chapters, boosting is developed on weak learners

**We use a subset of variables at each node split in case of Random Forest**

**Feedback :** Random Forest uses a subset of variables for node split such that a certain variable is not always used for node split.

**Additional Reading**

* The AdaBoost algorithm uses the [exponential loss function](https://mbernste.github.io/files/notes/AdaBoost.pdf), e−yh(x) to develop the expressions for the probability distribution and the weights assigned to the tree. You can go through the link mentioned for the derivation.

# AdaBoost Lab

The objective of this segment is to learn how to implement the AdaBoost algorithm in python. In this exercise, you have to go through the notebook attached to the page and answer the questions that follow.Refer to the [documentation](http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier) of AdaBoostClassifier if needed.

# base estimator: a weak learner with max\_depth=2

shallow\_tree = DecisionTreeClassifier(max\_depth=2, random\_state = 100)

# fit the shallow decision tree

shallow\_tree.fit(X\_train, y\_train)

# test error

y\_pred = shallow\_tree.predict(X\_test)

score = metrics.accuracy\_score(y\_test, y\_pred)

score -> ﻿

0.9385964912280702

You can try changing the number of trees in 'estimators = list(range(1, 50, 3))' to 'estimators = list(range(1, 200, 3))' and see if the accuracy increases. Note that we have used 'accuracy\_score' as the evaluation metric here. We can use other evaluation [metrics](http://scikit-learn.org/stable/modules/model_evaluation.html)also like 'roc\_auc\_curve'.

estimators = list(range(1, 50, 3))

abc\_scores = []

for n\_est in estimators:

ABC = AdaBoostClassifier( base\_estimator=shallow\_tree, n\_estimators = n\_est)

ABC.fit(X\_train, y\_train)

y\_pred = ABC.predict(X\_test)

score = metrics.accuracy\_score(y\_test, y\_pred)

abc\_scores.append(score)

|  |  |
| --- | --- |
| What is the accuracy of the initial weak learner we choose?  Top of Form  0.94  Feedback :  *The shallow decision tree we fit at the start gives an accuracy of 0.9385964912280702, which on rounding off gives 0.94.*  Bottom of Form | As we increase the number of estimators in the AdaBoostClassifier, what happens to the accuracy of the model?  **Increases**  Feedback :  *We see in the graph that as we increase the number of trees, the accuracy increases in a zigzag way* |

# **Understanding Gradient Boosting**

|  |  |
| --- | --- |
| The problem statement is same , we start with data points having feature vector Xi where we try to fit a function F(X), given Xi is the input it must produce a value which is approximately yi. This is the function we are trying to model by learning the function. | In the booting scheme of things we start with a crud Funtion Fo, may be this just predicts the mean of all the target variables. So, this crud function F0, given an Xi it just produces the mean value of y i.e. yavg.  Then for every Xi there is a gap, which is Yi – yavg. This gap is referred to as residue.  Then let’s say we try to produce a function F0 (X) + F1(X) given this feature which try to fill this gap. So essentially, we try to do a regularized regression fit in form of a function F1(X) which try to fill the gaps (residues).  The above formulation will work for strictly linear data points but for non-linear cases it wont produce good model. But if we try to fit the equation into such data points then the predicted yi would be far away from actual Yi.  So the gap is yi – yi0, where yi0 is my function F0 predicted.  Now we try build our 2nd model F1 which tries to fill this gap -> yi – yi0. After we fit the model F1 on the residuals yi−F0(xi), the prediction we get for xi is F1(xi). *The target variable for*F1*was*yi−F0(xi)*. Hence, the residual will be target variable - predicted value which is*yi−F0(xi)−F1(xi) |

Now, like we created a new function F1 and trained it on yi−F0(xi), we create a new function F2 as the next model to be trained in the Gradient Boosting Algorithm. What do you think F2 will train on?

yi−F0(xi)−F1(xi)

**Feedback :** *In every step/iteration of Gradient Boosting, we fit the new model on the residual of the corresponding step. So here*F2*will fit on the residual*yi−F0(xi)−F1(xi)*.*

Let us generalize the above expression for a model Ft. What is the general expression of the target variable the model Ft trains on? Assume, y is the initial target variable.

y**-**[Fo+F1**+**F2**+.........+**Ft−2**+**Ft−1]

**Feedback :** F1*trains on*y−F0*,*F2*trains on*y−F0−F1=y−[F0+F1]*, and so on.*

*In general,*Ft*trains on the residuals generated by the model*Ft−1*which is*y*- [*F0+F1*+*F2*+.........+*Ft−2*+*Ft−1*].*

# Gradient in Gradient Boosting

|  |  |
| --- | --- |
| The initial model predicts means of the target variable. The subsequent model trains on the residue of the previous mode. Idea is to predict as close as possible to actual value. | So like this we create a data set F2 where say for example for X1 data point the predicted value is y1−F0(x1)−F1(x1)  Here, "F0(Xi) + F1(Xi) = yi" is not an appropriate interpretation. Instead, the screenshot indicates that F0(Xi) + F1(Xi) is approximately equal to yi. With every additional iteration, the left hand side of the approximation comes closer and closer to the right hand side, i.e. yi. That is why F2, F3 and so on need to be calculated, until accuracy is achieved, to a degree deemed sufficient.  We keep doing this as long as the residues we get by using the ensemble is still more than what we wants to residues to be. This whole process precisely a gradient descent problem. |

Say, you are trying to train model to predict sale price of house (Actual value 200K).

F0 = 150K

then F1 will train on (200-150) 50K and may be predict 40K

then F2 will train on (50-40) 10K and may be predict 5K

Prediction = F0+F1+F2 = 150+40+5 = 195K

We got an intuition of how the gradient boosting process helps in reducing the error with each iteration. To summaries, the gradient boosting algorithm has two steps at each iteration:

1. Find the residual and fit a new model on the residual
2. Add the new model to the older model and continue the next iteration

Let’s now see how this can be seen as a gradient descent problem. But before you move on to the next video, let's have a quiz to reinforce your understanding of gradient descent. You have learned that in the gradient descent algorithm, a function G(x) decreases fastest around a point “a” if one goes along the negative gradient of G(x). It follows the iterative process:

an+1 = an- λ dG(x)/dx

Now if the function is G(x,y),  we have the following formulation:

xn+1 = xn- λ ∂G(x,y)/∂x

yn+1 = yn- λ ∂G(x,y)/∂y

Note that ∂/∂x refers to [the partial derivative](http://tutorial.math.lamar.edu/Classes/CalcIII/PartialDerivatives.aspx#PD_PartialD_Ex1a).

|  |  |
| --- | --- |
|  |  |
|  |  |

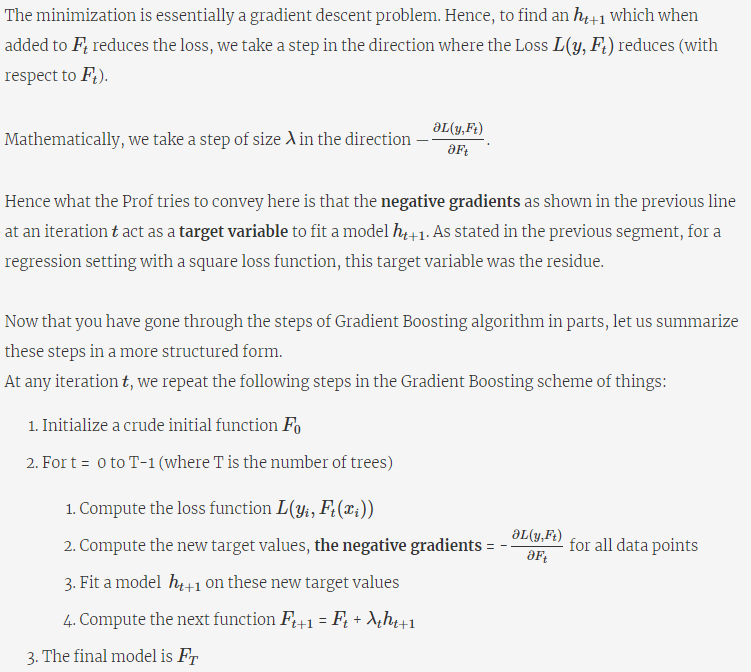
|  |  |
| --- | --- |
| In regular linear regression we try to minimize the square error in the cost function. In this case the closing in on the residue is equivalent to the gradient only because we are dealing with the square error of the loss function. Now from lasso (L1), ridge (L2) regression we know the there is a typical problem with the square error loss function that is they are overly sensitive to the outliers. | There are several situations where we don’t want to be sensitive to outliers. Those cases we may not prefer the square loss error loss function. In such scenarios we better off viewing it as a gradient and not as a residue.  So the strategy of closing in on the residues at each step can be seen as taking a **step towards the negative gradient of the loss function**. We can observe here that gradient descent helps in generalizing the boosting process.    Please **note** that this explanation and the aforementioned process of closing in on the residues is only valid for square error loss function. |

**Additional Reading**

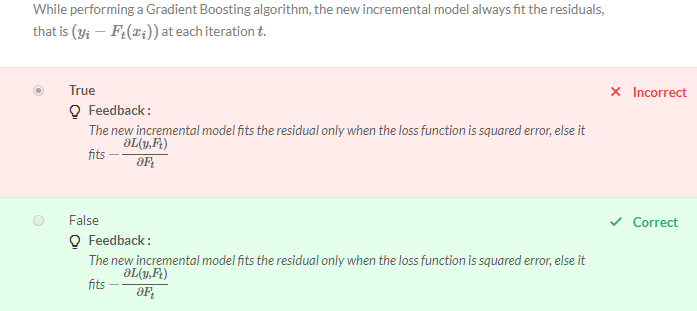
* The following [link](https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications) has the list of cost functions that can be used while solving a problem using Gradient Descent.

# Gradient Boosting Algorithm

|  |  |
| --- | --- |
|  | After a series of iteration F0 -> F1 -> F2 …. Ft -> Ft+1 .. we reach at a function FT which should be the close to actual yi. Even though FT leaves us with little gap (yi – Yipred), which we termed as loss function L. Now this loss function not a squared error loss function rather can be any generic function as long as it is differentiable. |



To explain this in just words, at each iteration we add an incremental model, which fits on the negative gradients of the loss function evaluated at current target values. This incremental model can be a linear regression model or a decision tree or any other model. We stop when we see that the gradients are very close to zero. For a regression setting, this means that when the residuals are very close to zero, we stop iterating Here, λt, known as the **learning rate,**is typically between 0 and 1.



**Comprehension - Loss Functions in Gradient Boosting**

We can use different loss functions depending upon the kind of model we wish to build. We have seen that AdaBoost uses the exponential loss function. We have also seen that squared error loss functions (for regression) are very sensitive to outliers, as explained by the professor earlier. To tackle this problem, we can use alternate loss functions, such as the  **Huber loss criterion** defined as follows:

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| Negative sign is there in the output because, in Huber loss function, the sign before Ft is negative. If you take derivative of (y+Ft), there will be no negative sign at the beginning. But here Huber Loss function is (y-Ft) square and not (y+Ft) square. |  |

We have learned that in the gradient descent algorithm, a function G(x) decreases fastest around a point “a” if one goes along the negative gradient of G(x). It follows:

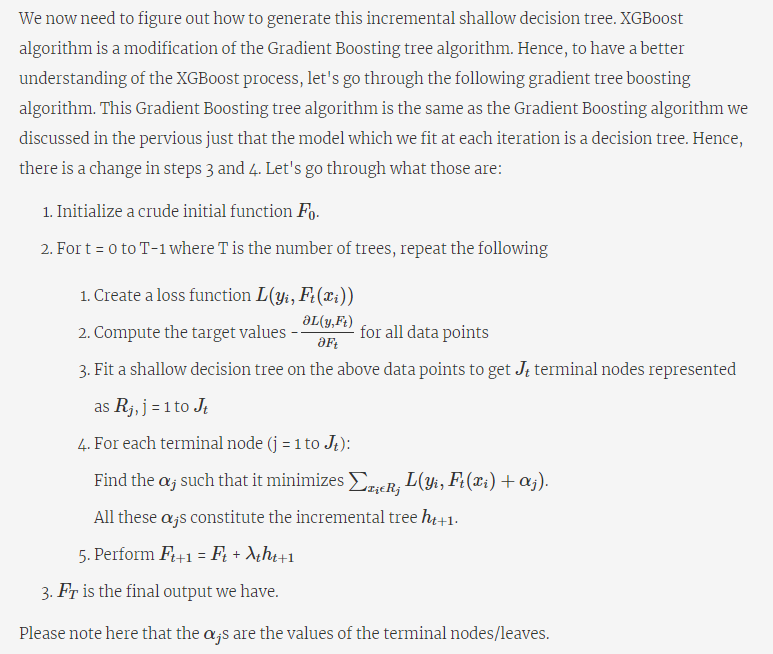
an+1 = an- λ dG(x)/dx

If we look at our function here, to form an analogy, G(x) is L(y,Ft) and x is Ft. Hence, this direction turns out to be −∂L(y,Ft)/∂Ft.

# XGBoost

XGBoost was first developed by Taiqi Chen and became famous in solving the Higgs Boson problem. XG Boost is an abbreviation for **extreme gradient boosting**.

Let us first recall how regression tree works. Eachleaf of a regression tree predicts a numerical value y for a data point x, i.e. when you push a data point x through the tree, it ends up in one of the leaves and takes the value corresponding to that leaf. XGBoost stands for Extreme Gradient Boosting which involves Gradient Boosting on shallow decision trees. Like before, we add a shallow decision tree to a previous model to create a new model.



The stopping criteria here is that the gradients are close to zero (or exactly zero). We can see that Ft is a tree created at the end of each iteration by adding an incremental tree ht to Ft−1.

# The mathematics behind the XGBoost model:

Extreme Gradient Boosting (XGBoost) is similar to gradient boosting framework but more efficient and advanced implementation of Gradient Boosting algorithm. Both XGBoost and GBM follow the principle of gradient boosted trees, but XGBoost uses a more regularised model formulation to control overfitting, which gives it better performance, which is why it’s also known as ‘regularised boosting‘ technique.

In an ideal machine learning model, the objective function is a sum of Loss function “L” and regularization omega “Ω”. Loss function controls the predictive power of the algorithm and regularization controls its simplicity.

**Objective Function : Training Loss + Regularization,** The above Gradient Boosting Tree algorithm has the objective function as only the Training Loss while the XGBoost objective function constitutes of loss function evaluated over, all predictions and sum of regularization term for all predictors ('T' trees).

|  |  |
| --- | --- |
|  | Where ht means predictions coming from the tth tree. |

The major difference between the Gradient Boosting and the XGBoost is that XGBoost incorporates the regularization parameter in its objective function to control over-fitting. *Though both 'XGBoost' and 'gbm' follows the principle of gradient boosting they differ in modelling details. As xgboost uses a more regularized model formalisation to maintains the simplicity of the model which results in better performance.*

**Why do we not use conventional gradient descent for the optimization of the Cost function in XGBoost?**

Top of Form

**Cost function includes “functions” as parameters and cannot be optimized using traditional optimization methods like gradient descent in Euclidean space**

**Feedback :**

*Yes, we use additive training or second order optimization technique to deal with such scenarios.*

Bottom of Form

 After defining the objective function the next challenge is to optimize the parameters. What we need to learn are those functions ht which contain the structure of the tree and leaf scores, which seems fairly complex optimization problem to be dealt with “Gradient descent” technique as this ensemble model includes “functions” as parameters and can’t be optimized using conventional method. Also, it’s not easy to train all trees at once, hence the “**Additive Training**” method is being used in XGBoost.

# Overview of stacking

Stacking mainly differ from bagging and boosting on two points. First stacking often considers **heterogeneous weak learners** (different learning algorithms are combined) whereas bagging and boosting consider mainly homogeneous weak learners. Second, stacking learns to combine the base models using a meta-model whereas bagging and boosting combine weak learners following deterministic algorithms.

# Stacking

As we already mentioned, the idea of stacking is to learn several different weak learners and **combine them by training a meta-model** to output predictions based on the multiple predictions returned by these weak models. So, we need to define two things in order to build our stacking model: the L learners we want to fit and the meta-model that combines them.

For example, for a classification problem, we can choose as weak learners a KNN classifier, a logistic regression and a SVM, and decide to learn a neural network as meta-model. Then, the neural network will take as inputs the outputs of our three weak learners and will learn to return final predictions based on it.

So, assume that we want to fit a stacking ensemble composed of L weak learners. Then we have to follow the steps thereafter:

* split the training data in two folds
* choose L weak learners and fit them to data of the first fold
* for each of the L weak learners, make predictions for observations in the second fold
* fit the meta-model on the second fold, using predictions made by the weak learners as inputs

In the previous steps, we split the dataset in two folds because predictions on data that have been used for the training of the weak learners are **not relevant for the training of the meta-model**. Thus, an obvious drawback of this split of our dataset in two parts is that we only have half of the data to train the base models and half of the data to train the meta-model. In order to overcome this limitation, we can however follow some kind of “k-fold cross-training” approach (similar to what is done in k-fold cross-validation) such that all the observations can be used to train the meta-model: for any observation, the prediction of the weak learners are done with instances of these weak learners trained on the k-1 folds that do not contain the considered observation. In other words, it consists in training on k-1 fold in order to make predictions on the remaining fold and that iteratively so that to obtain predictions for observations in any folds. Doing so, we can produce relevant predictions for each observation of our dataset and then train our meta-model on all these predictions.