Ensemble method

Bias: Bias describes how well a model matches the training set. A model with high bias won't match the data set closely, while a model with low bias will match the data set very closely. Bias comes from models that are overly simple and fail to capture the trends present in the data set

In his 1980 paper entitled “The need for bias in learning generalizations”, Tom Mitchell introduced the first use of the word “bias” in machine learning. He defined it to mean that a learning algorithm will not generalize unless it introduces some form of preference or restriction over the space of possible functions. Without any limitation or preference, the learning algorithm can memorize any data set without generalizing. This was later formalized in terms of the VC dimension (for a fixed-complexity function space), the No Free Lunch theorem, and structural risk minimization (for nested families of function spaces of increasing complexity.

This use of “bias” is closely related to the bias-variance tradeoff, because a learning algorithm with no bias (in the Mitchell sense) will have low bias and high variance in the bias-variance sense.

The most common interpretation of bias is with regards to the [bias–variance tradeoff](https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff). Essentially, bias here is a source of error in your model that causes it to over-generalize and underfit your data. In contrast, variance is sensitivity to noise in the data that causes your model to overfit. We call it a tradeoff because improving one will often make the other metric worse

**Introduction**

“Unity is strength”. This old saying expresses pretty well the underlying idea that rules the very powerful “ensemble methods” in machine learning. Roughly, ensemble learning methods, that often trust the top rankings of many machine learning competitions (including Kaggle’s competitions), are based on the hypothesis that combining multiple models together can often produce a much more powerful model.

We will discuss some well-known notions such as boostrapping, bagging, random forest, boosting, stacking and many others that are the basis of ensemble learning.

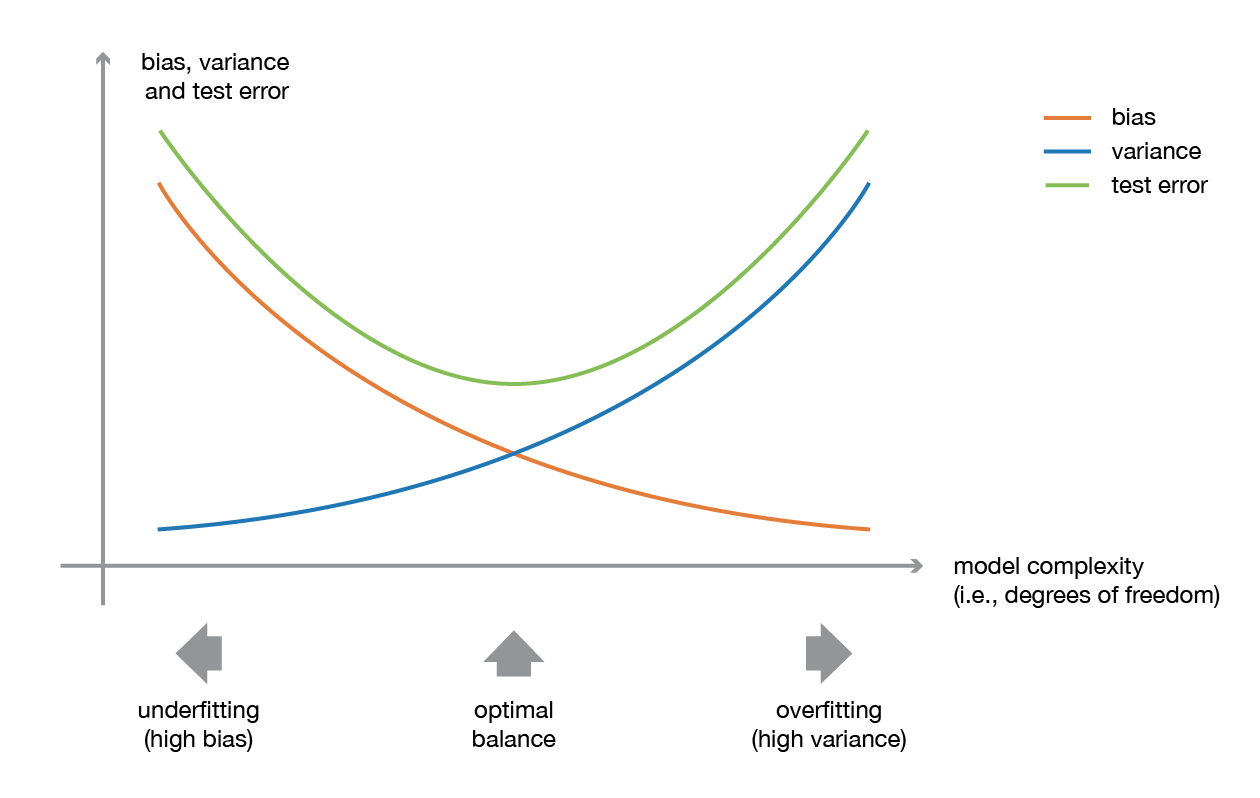
# What are ensemble methods?

Ensemble learning is a machine learning paradigm/model /protocol /example where multiple models (often called “weak learners”) are trained to solve the same problem and combined to get better results. The main hypothesis is that when weak models are correctly combined, we can obtain more accurate and/or robust models.

**Single weak learner:**

In machine learning, no matter if we are facing a classification or a regression problem, the choice of the model is extremely important to have any chance to obtain good results. This choice can depend on many variables of the problem: quantity of data, dimensionality of the space, distribution hypothesis…

A low bias and a low variance, although they most often vary in opposite directions, are the two most fundamental features expected for a model. Indeed, to be able to “solve” a problem, we want our model to have enough degrees of freedom to resolve the underlying complexity of the data we are working with, but we also want it to have not too much degrees of freedom to avoid high variance and be more robust. This is the well known **bias-variance tradeoff**.



In ensemble learning theory, we call **weak learners** (or **base models**) models that can be used as building blocks for designing more complex models by combining several of them. Most of the time, these basics models perform not so well by themselves either because they have a high bias (low degree of freedom models, for example) or because they have too much variance to be robust (high degree of freedom models, for example). Then, the idea of ensemble methods is to try reducing bias and/or variance of such weak learners by combining several of them together in order to create a **strong learner** (or **ensemble model**) that achieves better performances.

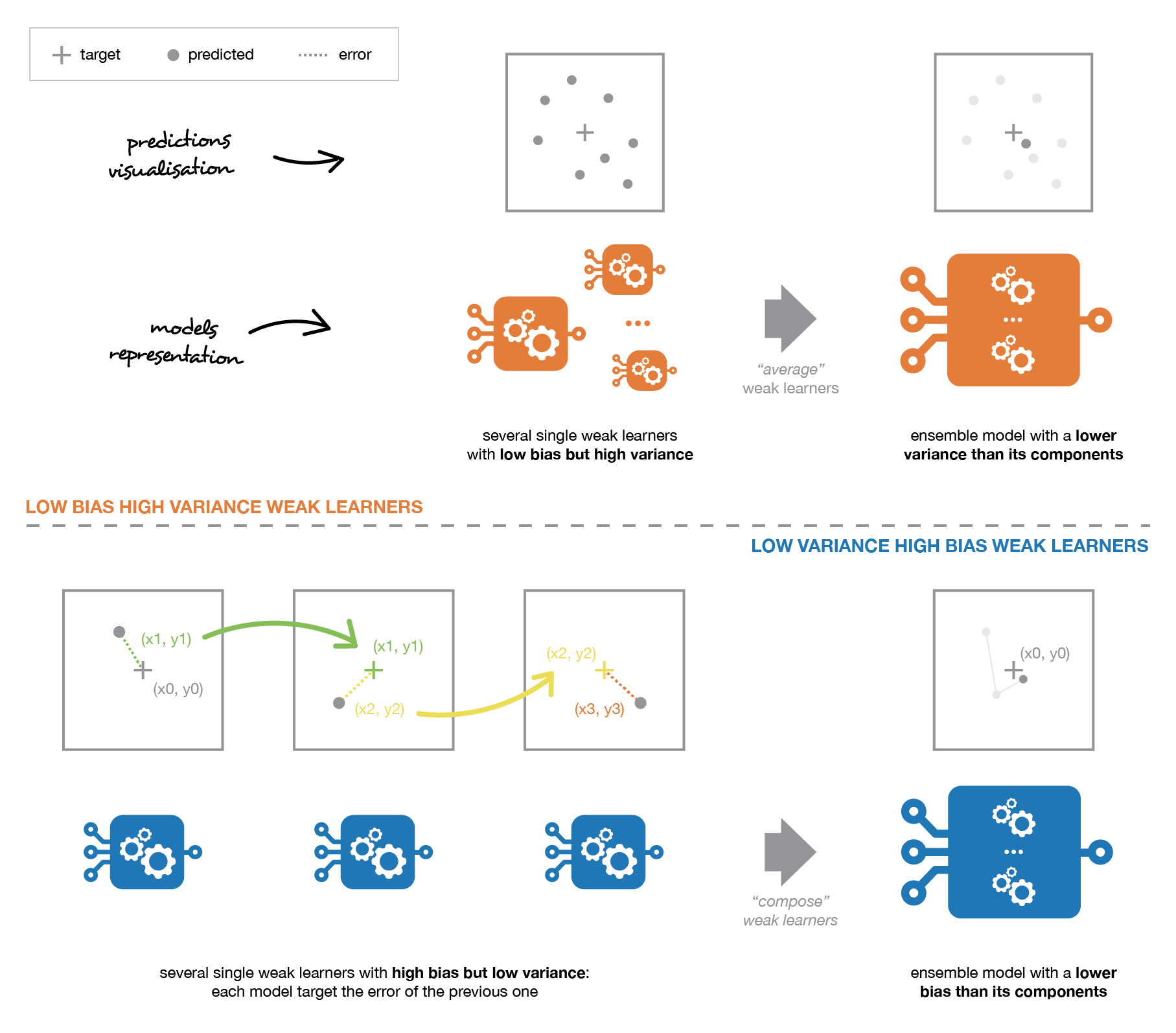
**Combined model :**

One important point is that our choice of weak learners should be **coherent with the way we aggregate these models**. If we choose base models with low bias but high variance, it should be with an aggregating method that tends to reduce variance whereas if we choose base models with low variance but high bias, it should be with an aggregating method that tends to reduce bias.

This brings us to the question of how to combine these models. We can mention three major kinds of meta-algorithms that aims at combining weak learners:

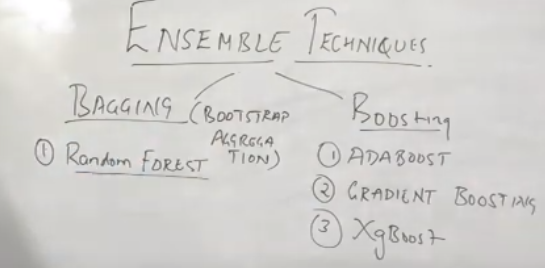
* **bagging**, that often considers homogeneous weak learners, learns them independently from each other in parallel and combines them following some kind of deterministic averaging process
* **boosting**, that often considers homogeneous weak learners, learns them sequentially in a very adaptative way (a base model depends on the previous ones) and combines them following a deterministic strategy
* **stacking**, that often considers heterogeneous weak learners, learns them in parallel and combines them by training a meta-model to output a prediction based on the different weak models predictions

Very roughly, we can say that bagging will mainly focus at getting an ensemble model with less variance than its components whereas boosting and stacking will mainly try to produce strong models less biased than their components (even if variance can also be reduced).



Weak learners can be combined to get a model with better performances. The way to combine base models should be adapted to their types. Low bias and high variance weak models should be combined in a way that makes the strong model more robust whereas low variance and high bias base models better be combined in a way that makes the ensemble model less biased.

<https://www.youtube.com/watch?v=KIOeZ5cFZ50>



**Bagging ( Bootstrap aggregation ):**

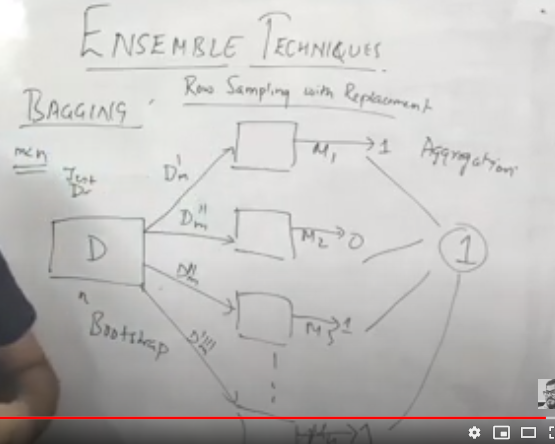
Suppose we have a dataset having N number of records and now are doing row sampling with replacement , which means we will be making sample of main dataset set with replacement having M number of rows . Here records/ rows present in first sample might also present in second sample or third sample.

Number of rows in bootstrap sample is less the number of row in main dataset

M<N

This method of sampling is known as bootstrap sampling .

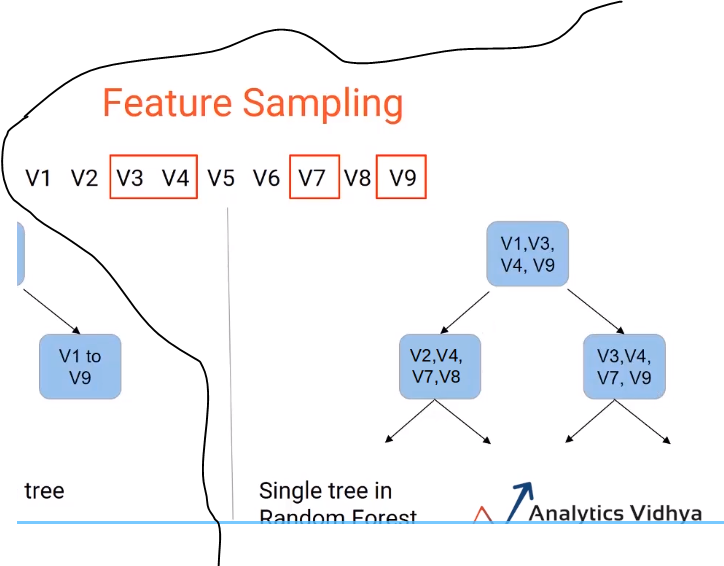
Bagging model: If we have 10 number of base models than 10 bootstrap sample dataset will be created and all base models will be trained and final prediction will be taken on voting or on base of majority count of class (classification prob) and aggregate/average value of regression problem.



## **Random forests**

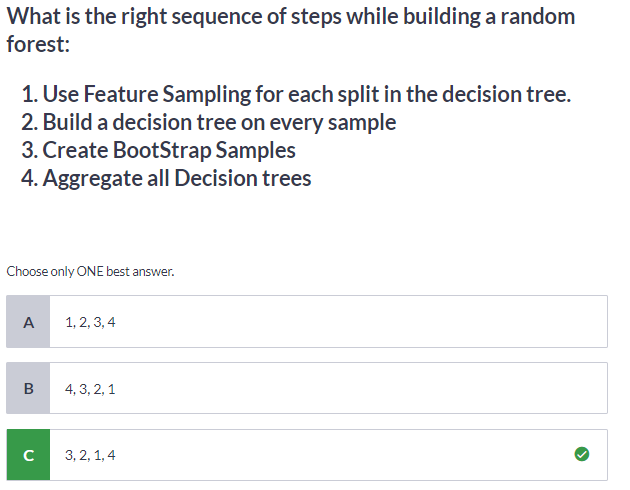
Learning trees are very popular base models for ensemble methods. Strong learners composed of multiple trees can be called “forests”. Trees that compose a forest can be chosen to be either shallow (few depths) or deep (lot of depths, if not fully grown). Shallow trees have less variance but higher bias and then will be better choice for sequential methods that we will described thereafter. Deep trees, on the other side, have low bias but high variance and, so, are relevant choices for bagging method that is mainly focused at reducing variance.

The **random forest** approach is a bagging method where **deep trees**, fitted on bootstrap samples, are combined to produce an output with lower variance. However, random forests also use another trick to make the multiple fitted trees a bit less correlated with each others: when growing each tree, instead of only sampling over the observations in the dataset to generate a bootstrap sample, we also **sample over features** and keep only a random subset of them to build the tree.

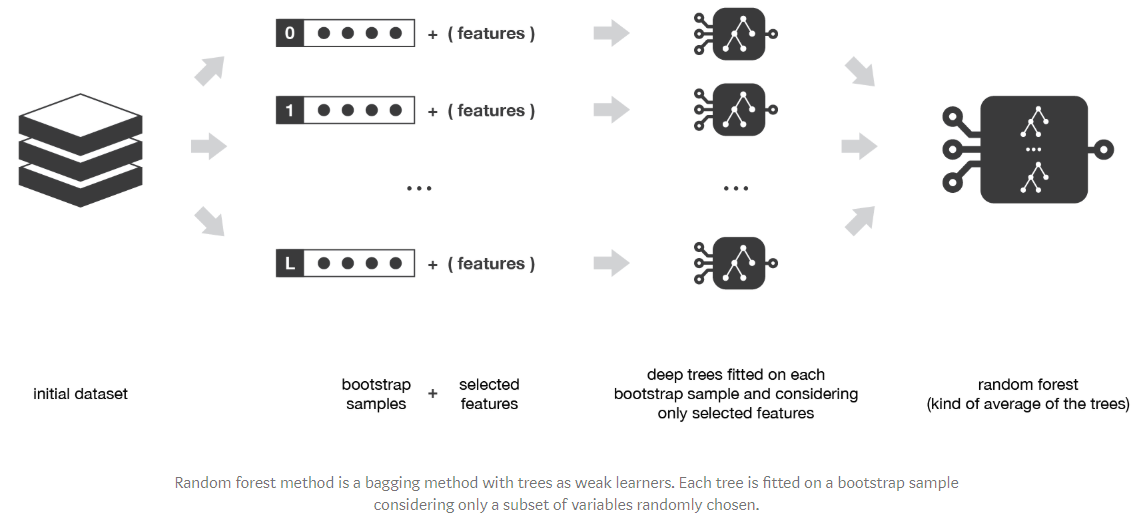


Let’s say we have create subset data for root node for best split and we will be creating different subset data at each node for finding best split and this process on for every node in every tree.

Feature sampling : Random features selection at each node slip like showed in above picture



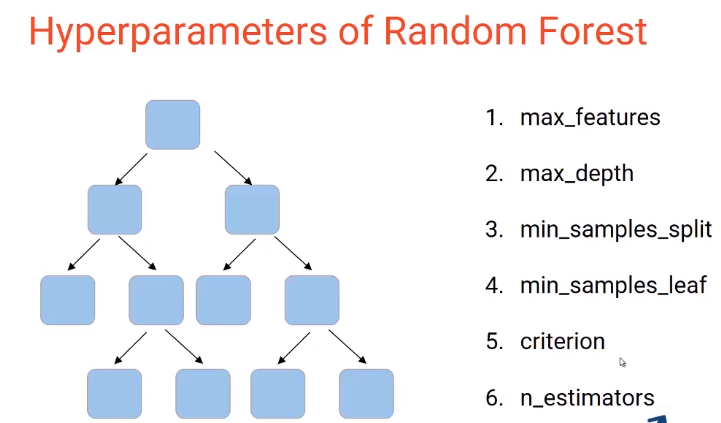
Sampling over features has indeed the effect that all trees do not look at the exact same information to make their decisions and, so, it reduces the correlation between the different returned outputs. Another advantage of sampling over the features is that**it makes the decision making process more robust to missing data**: observations (from the training dataset or not) with missing data can still be regressed or classified based on the trees that take into account only features where data are not missing. Thus, random forest algorithm combines the concepts of bagging and random feature subspace selection to create more robust models.



## **Random Subsets of features for splitting nodes**

The other main concept in the random forest is that only a [subset of all the features are considered for splitting each node](https://sebastianraschka.com/faq/docs/random-forest-feature-subsets.html) in each decision tree. Generally this is set to sqrt(n\_features) for classification meaning that if there are 16 features, at each node in each tree, only 4 random features will be considered for splitting the node. (The random forest can also be trained considering all the features at every node as is common in regression. These options can be controlled in the [Scikit-Learn Random Forest implementation](http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html" \t "_blank)).

The random forest combines hundreds or thousands of decision trees, trains each one on a slightly different set of the observations, splitting nodes in each tree considering a limited number of the features. The final predictions of the random forest are made by averaging the predictions of each individual tree



Max\_feature : No of feature in root node . Sqrt of total number of tress or logarithmic of total no of feature .

Max\_depth : Depth of the tree

Min\_sample\_split : Minimum amount of sample should present in node before slip , further split of tree stops if number of sample is less than that mentioned .

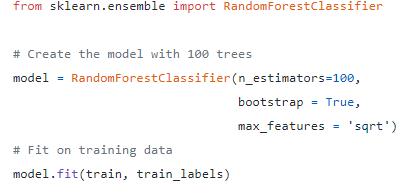
Min \_sample\_leaf : Minimum amount of sample present in leaf node , if sample amount is less than mentioned then that leaf will be ignored .

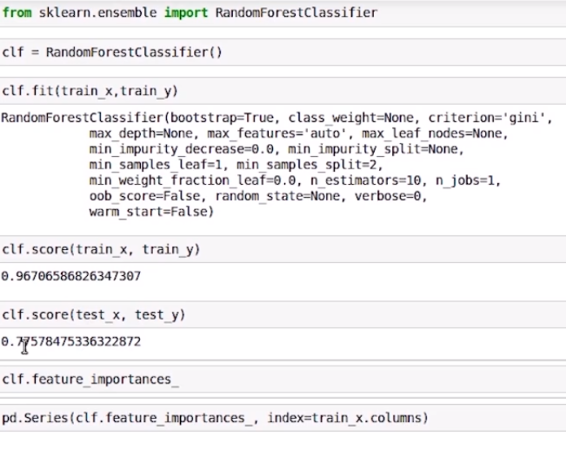
Criteria : Gini or Entropy

N\_estimator : number of tress

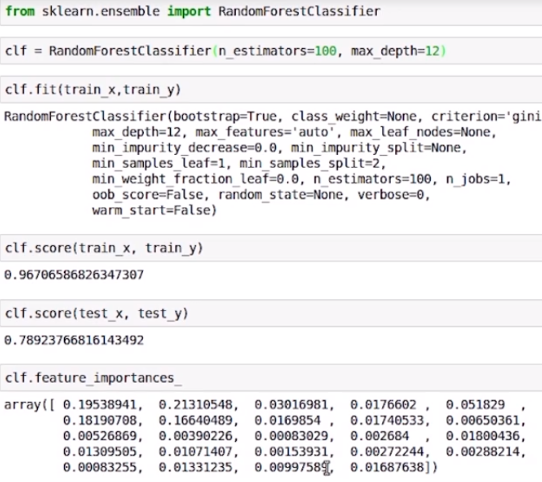
n\_jobs : Defines how many cores of system can random classifier can use

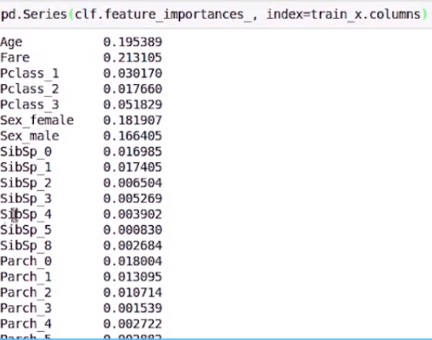
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>





With change in parameter we got improvement in accuracy in text data





# Focus on boosting :

In **sequential methods** the different combined weak models are no longer fitted independently from each others. The idea is to fit models **iteratively** such that the training of model at a given step depends on the models fitted at the previous steps. “Boosting” is the most famous of these approaches and it produces an ensemble model that is in general less biased than the weak learners that compose it.

## **Boosting :**

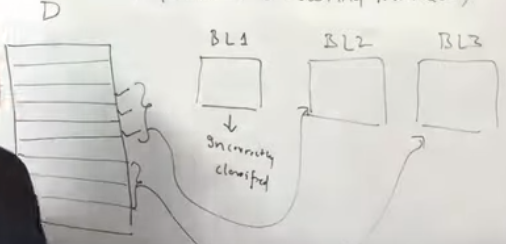
Boosting methods work in the same spirit as bagging methods: we build a family of models that are aggregated to obtain a strong learner that performs better. However, unlike bagging that mainly aims at reducing variance, boosting is a technique that consists in fitting sequentially multiple weak learners 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 parametrised). 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.

**Sampling in boosting method:** Suppose we have Dataset D having N no of rows and in boosting method

initially few rows from the dataset are randomly sampled and are feed to first model .

The wrongly classified rows as a sample is feed to next consecutive model .



## **Adaptative boosting**

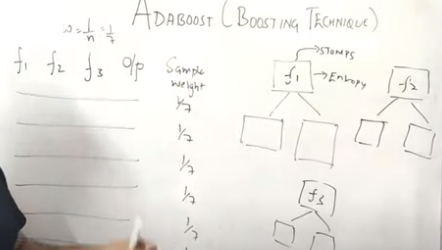
In adaptative boosting (often called “adaboost”), we try to define our ensemble model as a weighted sum of L weak learners .

Let’s began taking example : Suppose I have dataset of features f1, f2 f3 and O/P

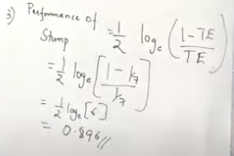
Step 1) We will assign sample weights W to each row of feature (W = 1/N)

2) We will be having our base model decision tree of one depth and model will be trained on this randomly sampled data . This decision trees are basically called as stumps

3) For each feature f1 , f2 ,f3 will create stump model and measure Entropy and Information gain . Model with less Entropy will be considered as our first base model



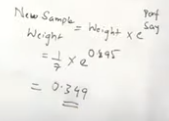
4) Calculating total error : summing up the weights of wrongly classified observations. Total error (TE) =1\7

5) Performance of stump (tree) = 

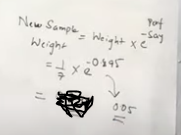
6) We need to update weight of sample using performance value

7) We need to increase the weights of wrongly classified observation and decrease the weights of properly classified observation .

This formulae for updating weights for incorrectly classified observation .

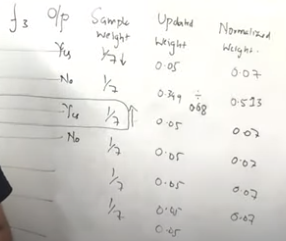


And for updating weights for correctly classified observation

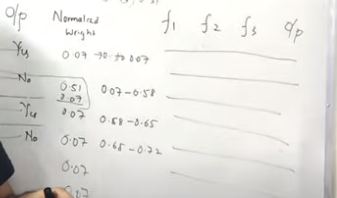


Here we can see that value of updated weight are decreasing for properly classified observations and increasing for wrongly classified observation .So that while creating sample for next model the probability of having wrongly classified observations are more compared

4) Normalize the updated weights : But these weights are not normalized that is their sum is not equal to one. To do this, we will sum them and divide each final weight with that sum.



5) After that, we want to make a second weak model. But to do that, we need a sample dataset on which the second weak model can be run. For making it, we will run N number of iterations. On each iteration, it will calculate a random number ranging between 0-1 and this random will be compared with class intervals we created and on which class interval it lies, that row will be selected for sample data set. So new sample data set would also be of N observation



6) This whole process will continue for M decision stumps. The final sequential tree would be considered as the final tree.

7) Voting of all the individual weak learners taken and that’s considered as final prediction Adaboost model

Classification model:<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier>

class sklearn.ensemble.AdaBoostClassifier(base\_estimator=None, n\_estimators=50, learning\_rate=1.0, algorithm='SAMME.R', random\_state=None)

**>>> from** **sklearn.model\_selection** **import** cross\_val\_score

**>>> from** **sklearn.datasets** **import** load\_iris

**>>> from** **sklearn.ensemble** **import** AdaBoostClassifier

**>>>** X, y = load\_iris(return\_X\_y=**True**)

**>>>** clf = AdaBoostClassifier(n\_estimators=100)

**>>>** scores = cross\_val\_score(clf, X, y, cv=5)

**>>>** scores.mean()

0.9...

Regression model :<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html#sklearn.ensemble.AdaBoostRegressor>

class sklearn.ensemble.**AdaBoostRegressor**(base\_estimator=None, n\_estimators=50, learning\_rate=1.0, loss='linear', random\_state=None)

**from** **sklearn.ensemble** **import** AdaBoostRegressor

**from** **sklearn.datasets** **import** make\_regression

X, y = make\_regression(n\_features=4, n\_informative=2,

**...**  random\_state=0, shuffle=**False**)

regr = AdaBoostRegressor(random\_state=0, n\_estimators=100)

regr.fit(X, y)

AdaBoostRegressor(n\_estimators=100, random\_state=0)

regr.feature\_importances\_ # Return the feature importances (the higher, themore important the feature).

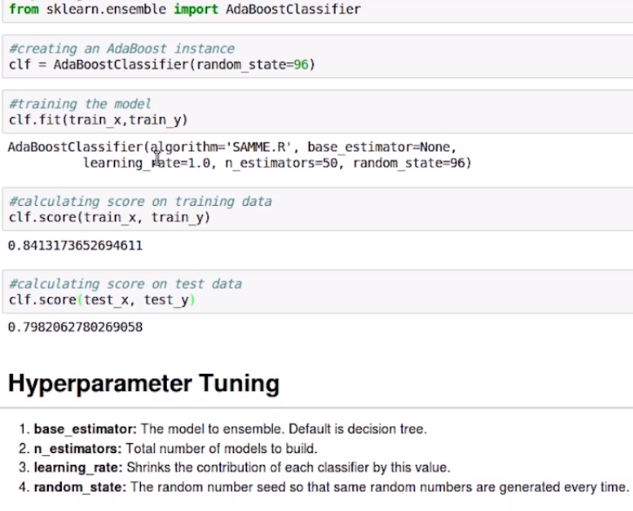
array([0.2788..., 0.7109..., 0.0065..., 0.0036...])

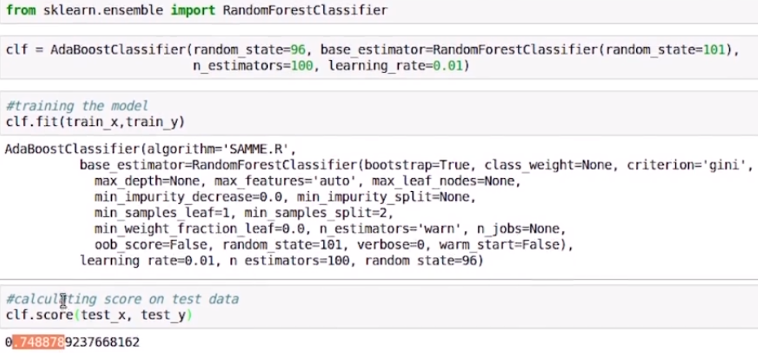
**>>>** regr.predict([[0, 0, 0, 0]])

array([4.7972...])

**>>>** regr.score(X, y)

0.9771..



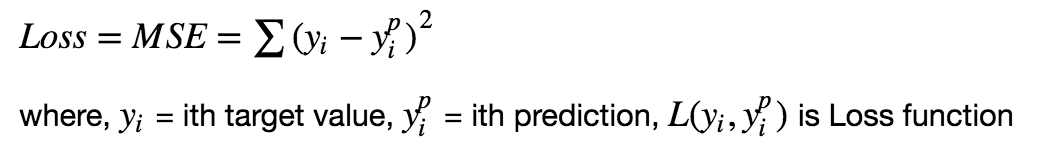


## **Gradient boosting:**

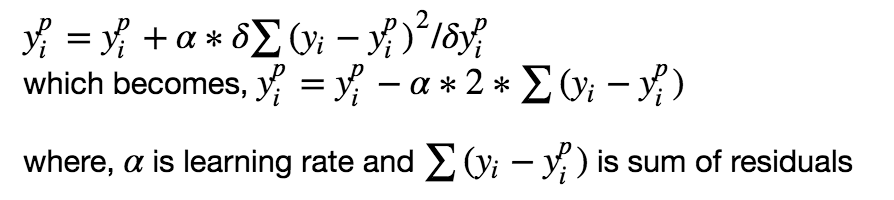
In gradient boosting, the ensemble model we try to build is also a weighted sum of weak learners.

*Gradient boosting 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.*(Wikipedia definition)

The objective of any supervised learning algorithm is to define a loss function and minimize it. Let’s see how maths work out for Gradient Boosting algorithm. Say we have **mean squared error (MSE) as loss** defined as:



We want our predictions, such that our loss function (MSE) is minimum. By using **gradient descent** and updating our predictions based on a learning rate, we can find the values where MSE is minimum.

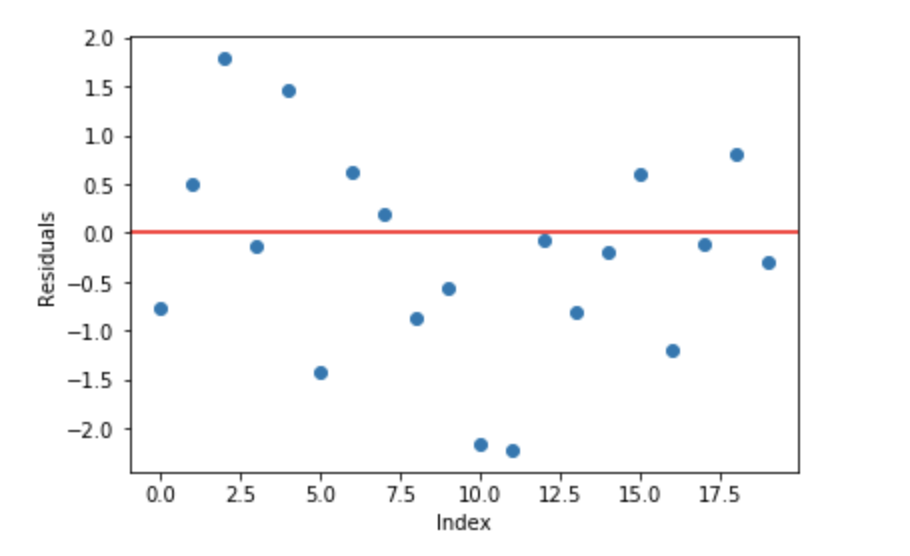


*So, we are basically updating the predictions such that the sum of our residuals is close to 0 (or minimum) and predicted values are sufficiently close to actual values.*

## **Intuition behind Gradient Boosting**

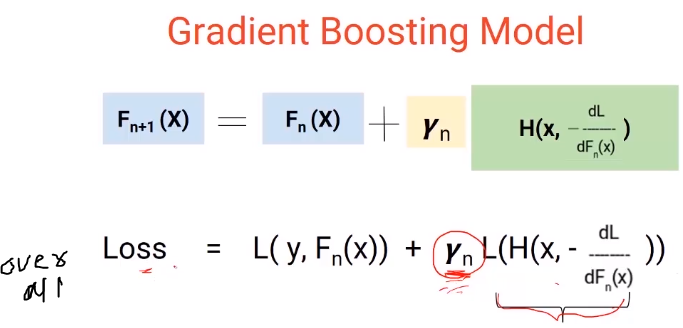
The logic behind gradient boosting is simple, (can be understood intuitively, without using mathematical notation)

A basic assumption of linear regression is that sum of its residuals is 0, i.e. the residuals should be spread randomly around zero.



Now think of these residuals as mistakes committed by our predictor model. Although, tree-based models (considering decision tree as base models for our gradient boosting here) are not based on such assumptions, but if we think logically (not statistically) about this assumption, **we might argue that, if we are able to see some pattern of residuals around 0, we can leverage that pattern to fit a model.**

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). Algorithmically, we are minimizing our loss function, such that test loss reach its minima.



## **Steps to fit a Gradient Boosting model**

*Let’s consider simulated data as shown in scatter plot below with 1 input (x) and 1 output (y) variables.*

1. Fit a simple linear regressor or decision tree on data (I have chosen decision tree in my code) **[call x as input and y as output]**
2. Calculate error residuals. Actual target value, minus predicted target value **[e1= y - y\_predicted1 ]**

3. Fit a new model on error residuals as target variable with same input variables **[call it e1\_predicted]**

4. Add the predicted residuals to the previous predictions  
 **[y\_predicted2 = y\_predicted1 + e1\_predicted]**

5. Fit another model on residuals that is still left. i.e. **[e2 = y - y\_predicted2]**and repeat steps 2 to 5 until it starts overfitting or the sum of become constant. Overfitting can be controlled by consistently checking accuracy on validation data.

[**GradientBoostingClassifier**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier) supports both binary and multi-class classification. The following example shows how to fit a gradient boosting classifier with 100 decision stumps as weak learners

class sklearn.ensemble.**GradientBoostingClassifier**(loss='deviance', learning\_rate=0.1, n\_estimators=100, subsample=1.0, criterion='friedman\_mse', min\_samples\_split=2, min\_samples\_leaf=1, min\_weight\_fraction\_leaf=0.0, max\_depth=3, min\_impurity\_decrease=0.0, min\_impurity\_split=None, init=None, random\_state=None, max\_features=None, verbose=0, max\_leaf\_nodes=None, warm\_start=False, presort='deprecated', validation\_fraction=0.1, n\_iter\_no\_change=None, tol=0.0001, ccp\_alpha=0.0)

clf = GradientBoostingClassifier(n\_estimators=100, learning\_rate=1.0,

**...**  max\_depth=1, random\_state=0)

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

clf.score(X\_test, y\_test)

0.913...

[**GradientBoostingRegressor**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ensemble.GradientBoostingRegressor) supports a number of [different loss functions](https://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting-loss) for regression which can be specified via the argument loss; the default loss function for regression is least squares ('ls')

est = GradientBoostingRegressor(n\_estimators=100, learning\_rate=0.1,

**...**  max\_depth=1, random\_state=0, loss='ls')

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

mean\_squared\_error(y\_test, est.predict(X\_test))

5.00...

**XGBoost ensemble method :**

The XGBoost works as same as gradient boosting but XGBoost is very faster than gradient boosting.

The beauty of this powerful algorithm lies in its scalability, which drives fast learning through parallel and distributed computing and offers efficient memory usage.

[XGBoost](https://xgboost.ai/)is a decision-tree-based ensemble Machine Learning algorithm that uses a [gradient boosting](https://en.wikipedia.org/wiki/Gradient_boosting) framework. In prediction problems involving unstructured data (images, text, etc.) artificial neural networks tend to outperform all other algorithms or frameworks. However, when it comes to small-to-medium structured/tabular data, decision tree based algorithms are considered best-in-class right now.

## Unique features of XGBoost

XGBoost is a popular implementation of gradient boosting. Let’s discuss some features of XGBoost that make it so interesting.

1. **Regularization**: It penalizes more complex models through both LASSO (L1) and Ridge (L2) [regularization](https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c) to prevent overfitting.
2. **Handling sparse data**: Missing values or data processing steps like one-hot encoding make data sparse. XGBoost incorporates a sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data.
3. **Weighted quantile sketch**: Most existing tree based algorithms can find the split points when the data points are of equal weights (using quantile sketch algorithm). However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data.

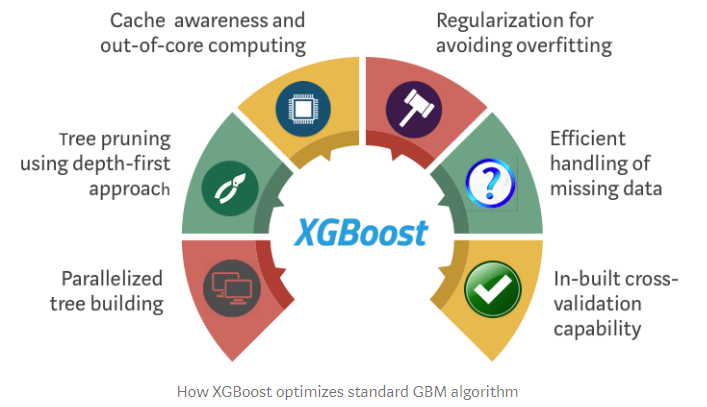
 XGBoost employs the distributed [weighted Quantile Sketch algorithm](https://arxiv.org/pdf/1603.02754.pdf) to effectively find the optimal split points among weighted datasets.

1. **Parallel processing** : Trees in XGBoost are built in sequentially, but implements parallel process of internal nodes hence its faster than Gradient boosting method .

Parallel node building at each induvial tree.

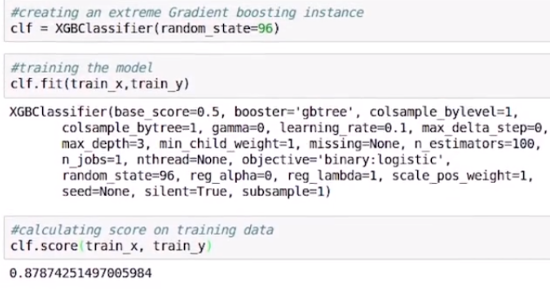
1. **Hardware Optimization**: This algorithm has been designed to make efficient use of hardware resources. This is accomplished by cache awareness by allocating internal buffers in each thread to store gradient statistics.
2. **Out-of-core computing**: This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory.
3. **Cross-validation**: The algorithm comes with built-in [cross-validation](https://towardsdatascience.com/cross-validation-in-machine-learning-72924a69872f) method at each iteration, taking away the need to explicitly program this search and to specify the exact number of boosting iterations required in a single run.

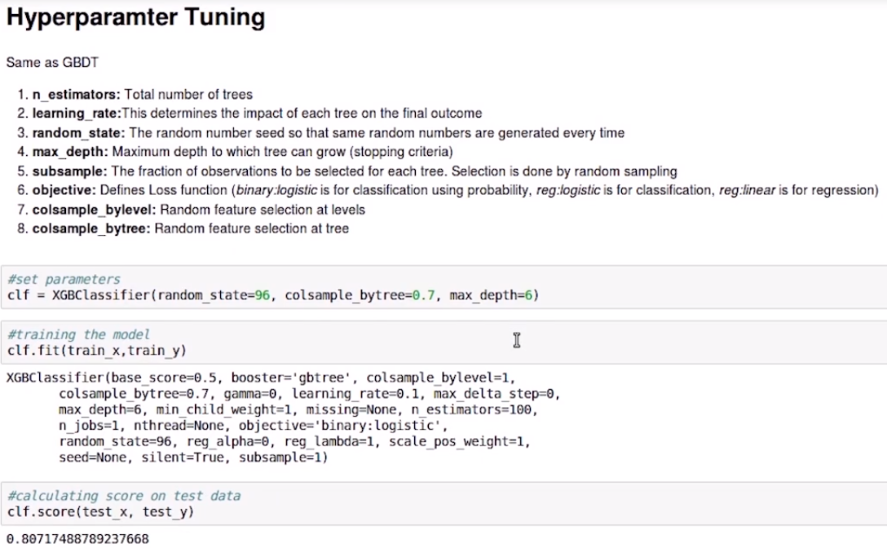
Parameters like the **number of trees or iterations**, the **rate at which the gradient boosting learns**, and the **depth of the tree**, could be optimally selected through **validation techniques like k-fold cross validation**. **Having a large number of trees might lead to overfitting. So, it is necessary to carefully choose the stopping criteria for boosting.**

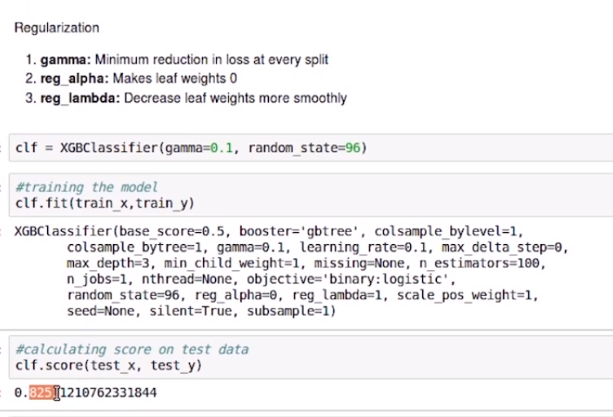


**Before starting building algorithm we need to install XGBoost from command prompt**

Pip install xgboost



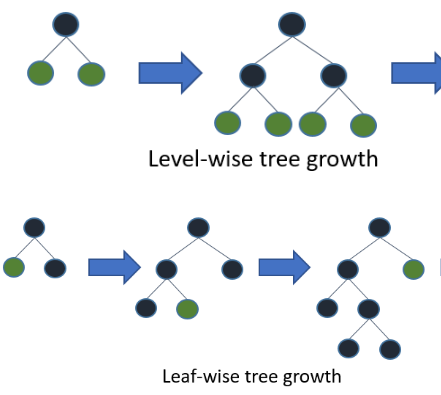




**Light GBM:**

Before discussing how Light GBM works, let’s first understand why we need this algorithm when we have so many others (like the ones we have seen above). Light GBM beats all the other algorithms when the dataset is extremely large. Compared to the other algorithms, Light GBM takes lesser time to run on a huge dataset.

LightGBM is a gradient boosting framework that uses tree-based algorithms and follows leaf-wise approach while other algorithms work in a level-wise approach pattern. The images below will help you understand the difference in a better way.



For classification

import lightgbm as lgb

train\_data=lgb.Dataset(x\_train,label=y\_train)

#define parameters

params = {'learning\_rate':0.001}

model= lgb.train(params, train\_data, 100)

y\_pred=model.predict(x\_test)

for i in range(0,185):

   if y\_pred[i]>=0.5:

   y\_pred[i]=1

else:

   y\_pred[i]=0

For regression :

import lightgbm as lgb

train\_data=lgb.Dataset(x\_train,label=y\_train)

params = {'learning\_rate':0.001}

model= lgb.train(params, train\_data, 100)

from sklearn.metrics import mean\_squared\_error

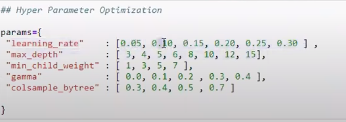
rmse=mean\_squared\_error(y\_pred,y\_test)\*\*0.5

**Parameters**

* num\_iterations:
  + It defines the number of boosting iterations to be performed.
* num\_leaves :
  + This parameter is used to set the number of leaves to be formed in a tree.
  + In case of Light GBM, since splitting takes place leaf-wise rather than depth-wise, num\_leaves must be smaller than 2^(max\_depth), otherwise, it may lead to overfitting.
* min\_data\_in\_leaf :
  + A very small value may cause overfitting.
  + It is also one of the most important parameters in dealing with overfitting.
* max\_depth:
  + It specifies the maximum depth or level up to which a tree can grow.
  + A very high value for this parameter can cause overfitting.
* bagging\_fraction:
  + It is used to specify the fraction of data to be used for each iteration.
  + This parameter is generally used to speed up the training.
* max\_bin :
  + Defines the max number of bins that feature values will be bucketed in.
  + A smaller value of max\_bin can save a lot of time as it buckets the feature values in discrete bins which is computationally inexpensive.

Hyper- parameter tuning

Defining parameters



Importing RandomizedSearchCv and GridSeachCv optimizers from sklearn.model\_selection



Defining model

Clasifier = xgboost.XGBostclassifier()

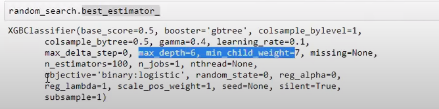
And fitting model and parameter in RandomizedsearchCV opimizer



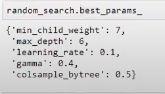
Training Randomize optimizer model with train data

Random\_search.fit(x,y)

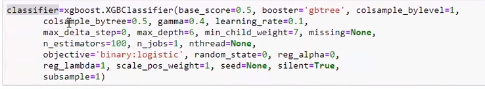
To know the best value of parameters selected by your model through Randomized optimizer with some other parameters suggest by model (default)



To know which are the best parameter from those give parameters to Randomizers.



We are create new model with the best parameter obtained through Randomizers .



Use cross validation and obtain different score for best parameter from RandomizerCV

Finally take mean score

