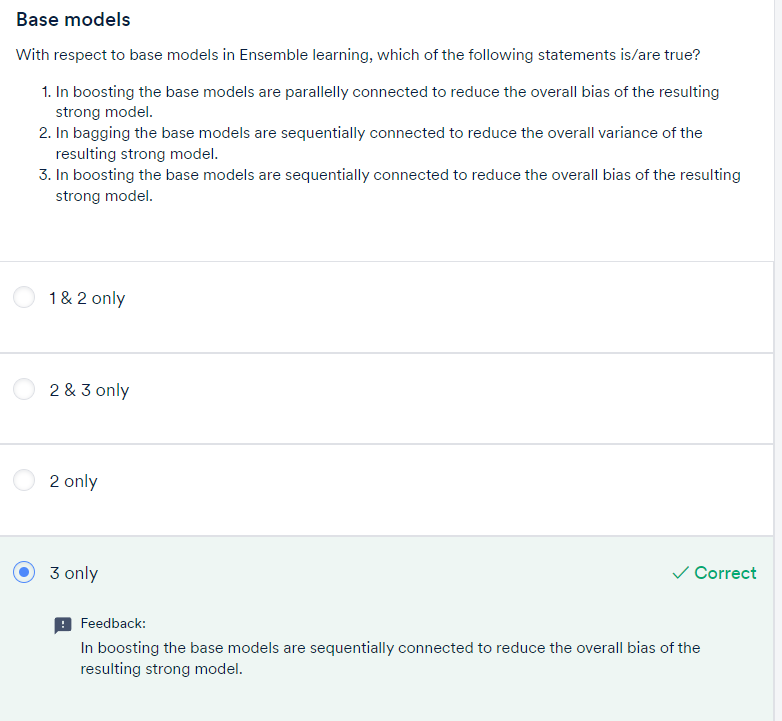
Boosting :

1. Adaptive Boosting
2. Gradient Boosting
3. XGBoost

This session will introduce you to the following topics:

* Ensemble models
* Introduction to boosting
* Building blocks of boosting
* AdaBoost procedure



# Weak Learners:

**To summarise:** Weak learners are combined sequentially such that each subsequent model corrects the mistakes of the previous model, resulting in a strong overall model that gives good predictions.

Through weak learners, you can do the following:

* Reduce the variance of the final model, making it more robust (generalisable)
* Train the ensemble quickly resulting in faster computation time

# 

# 

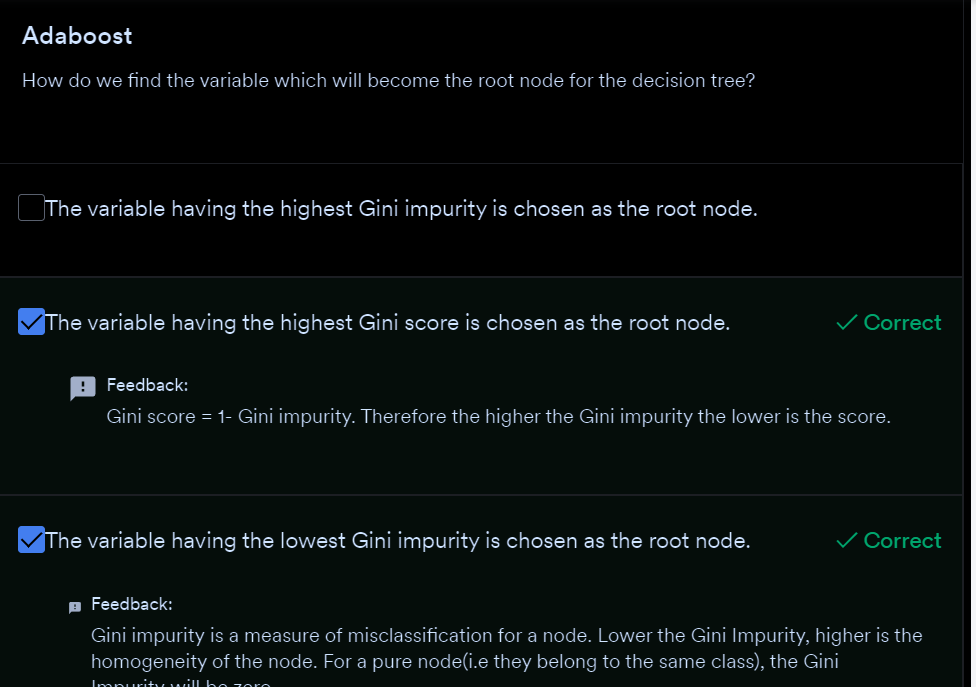
# Adaboost – Adaptive Boosting:

# 

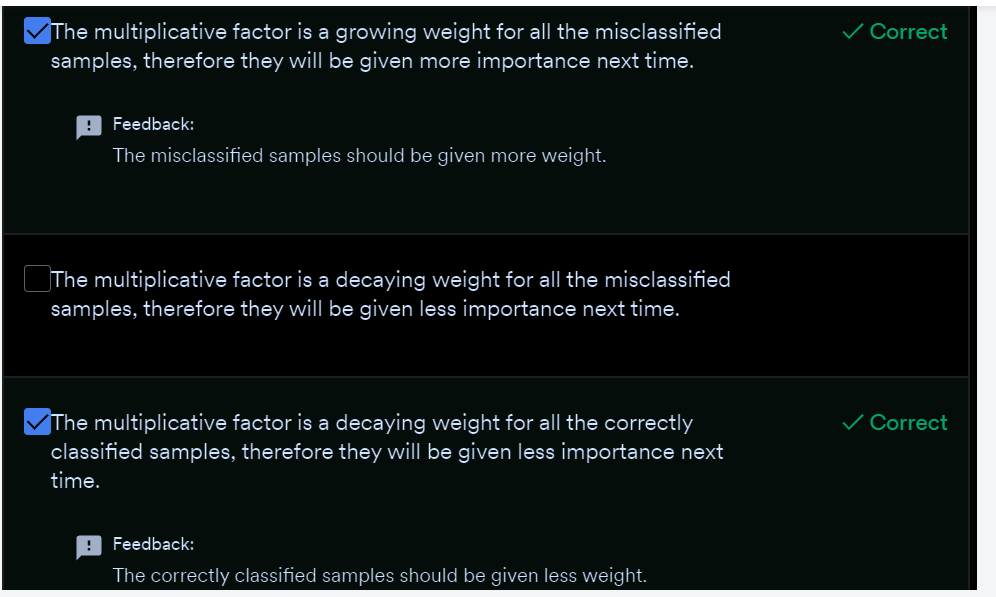
let’s see an overview of the steps that need to be taken in this boosting algorithm:

* AdaBoost starts with a uniform distribution of weights over training examples, i.e., it gives equal weights to all its observations. These weights tell the importance of each datapoint being considered.
* We start with a single weak learner to make the initial predictions.
* Once the initial predictions are made, patterns which were not captured by the previous weak learner are taken care of by the next weak learner by giving more weightage to the misclassified datapoints.
* Apart from giving weightage to each observation, the model also gives weightage to each weak learner. More the error in the weak learner, lesser is the weightage given to it. This helps when the ensembled model makes final predictions.
* After getting the two weights for the observations and the individual weak learners, the next weak learner in the sequence trains on the resampled data (data sampled according to the weights) to make the next prediction.
* The model will iteratively continue the steps mentioned above for a pre-specified number of weak learners.
* In the end, you need to take a weighted sum of the predictions from all these weak learners to get an overall strong learner.

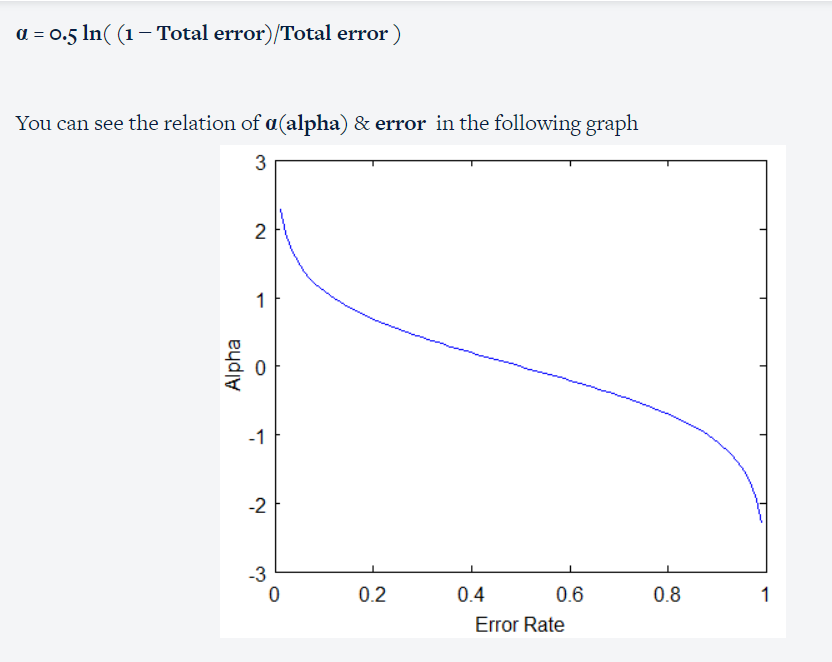
# 



Alpha = Amount of say

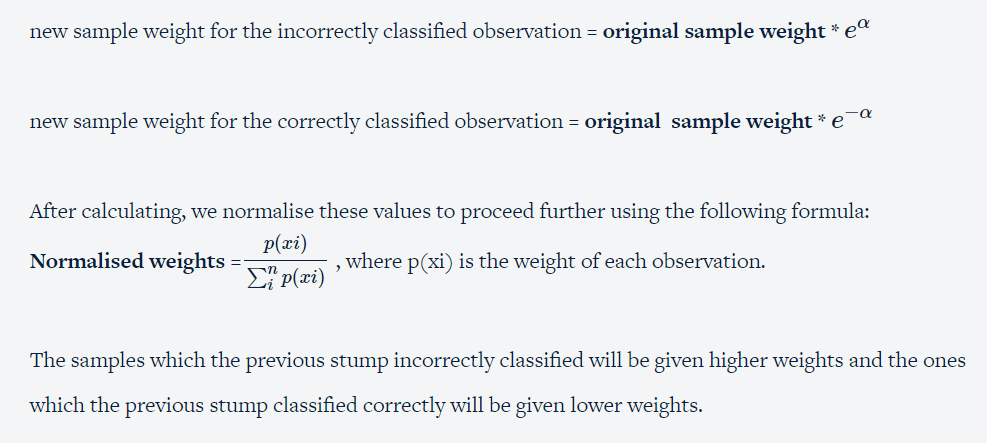


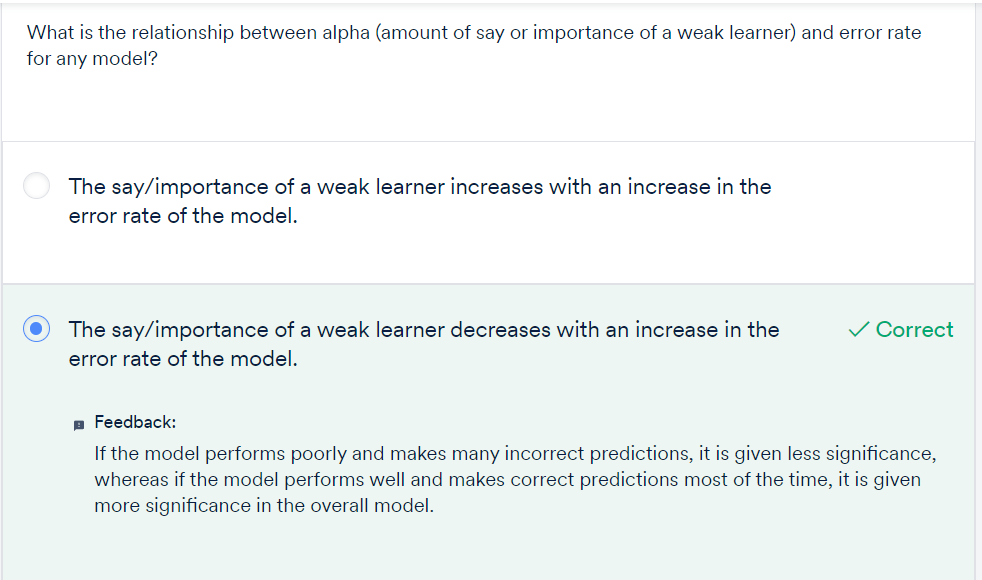
1. In AdaBoost, we start with a base model with equal weights given to every observation.
2. In the next step, the observations which are incorrectly classified will be given a higher weight so that when a new weak learner is trained, it will give more attention to these misclassified observations.
3. you get a series of models that have a different say according to the predictions each weak model has made
4. If the model performs poorly and makes many incorrect predictions, it is given less importance, whereas if the model performs well and makes correct predictions most of the time, it is given more importance in the overall model.



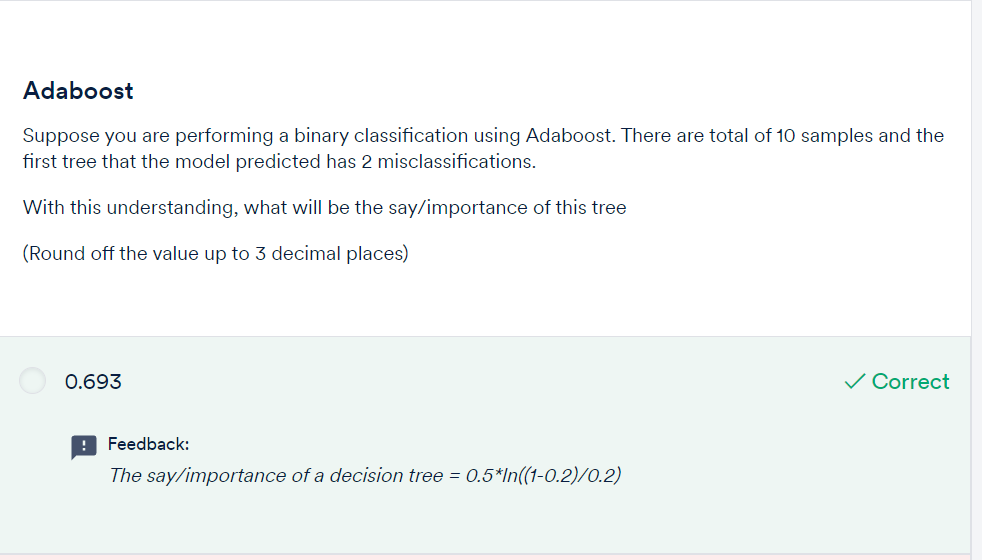
The value of the error rate lies between 0 and 1. So, let’s see how alpha and error is related.

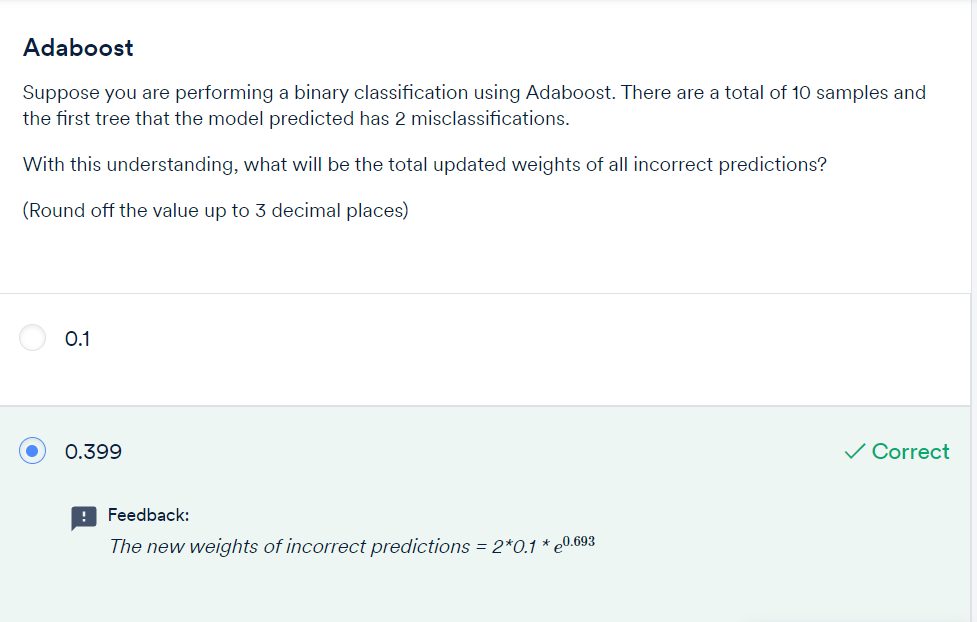
* When the base model performs with less error overall, then, as you can see in the plot above, the α is a large positive value, which means that the weak learner will have a high say in the final model.
* If the error is 0.5, it means that it is not sure of the decision, then the α = 0, i.e., the weak learner will have no say or significance in the final model.
* If the model produces large errors (i.e., close to 1), then α is a large negative value, meaning that the predictions it makes are incorrect most of the time. Hence, this weak learner will have a very low say in the final model.

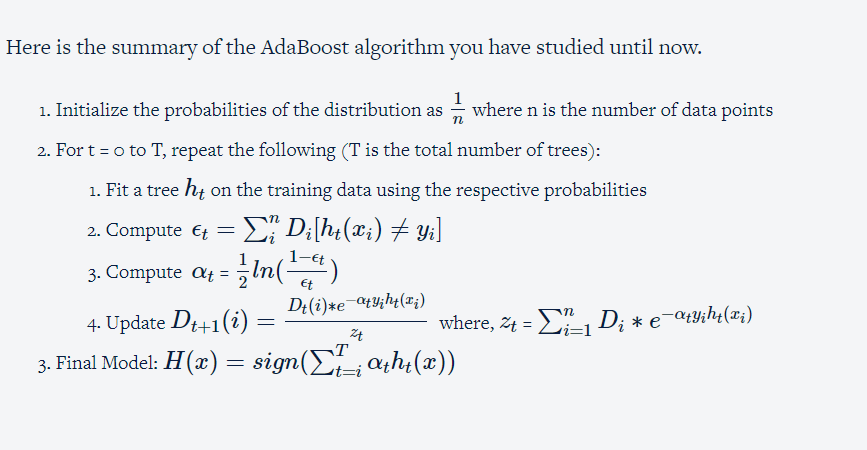


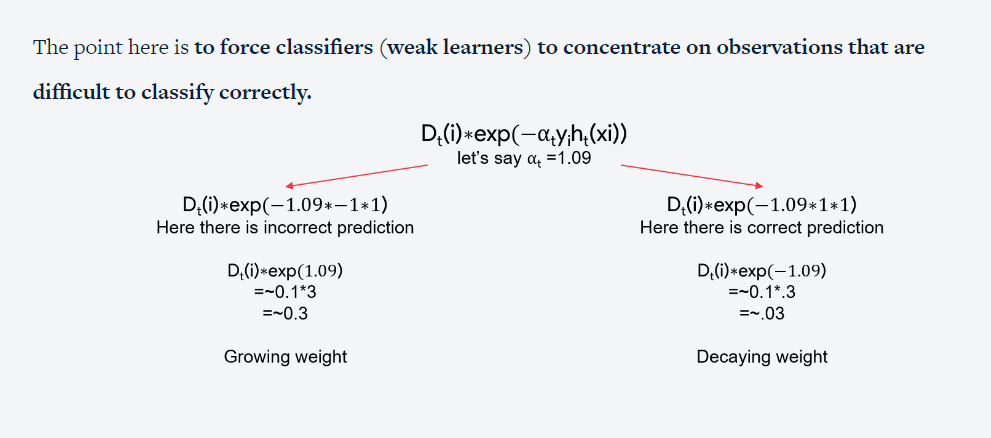


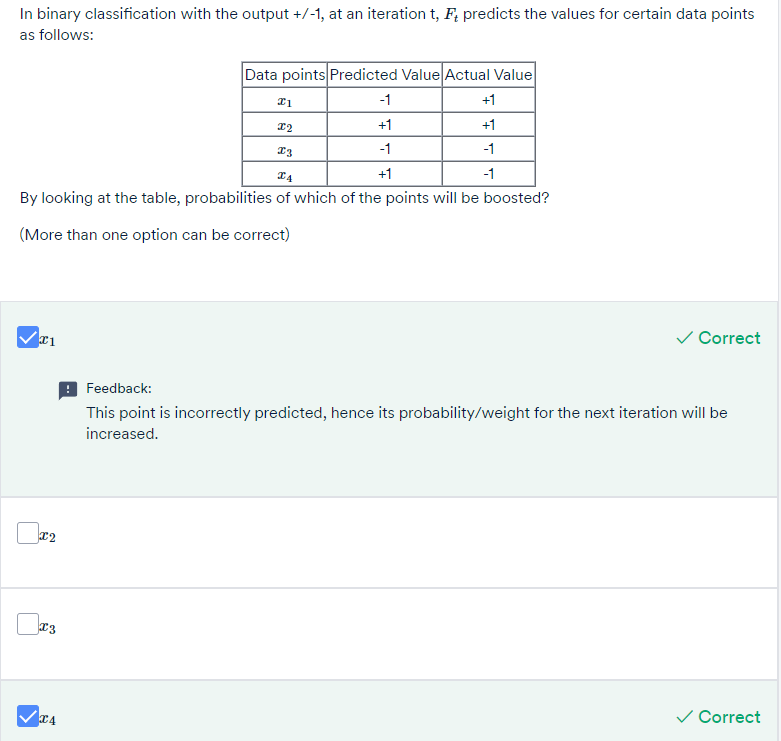
A tree with one node and two leaves is called **stump**





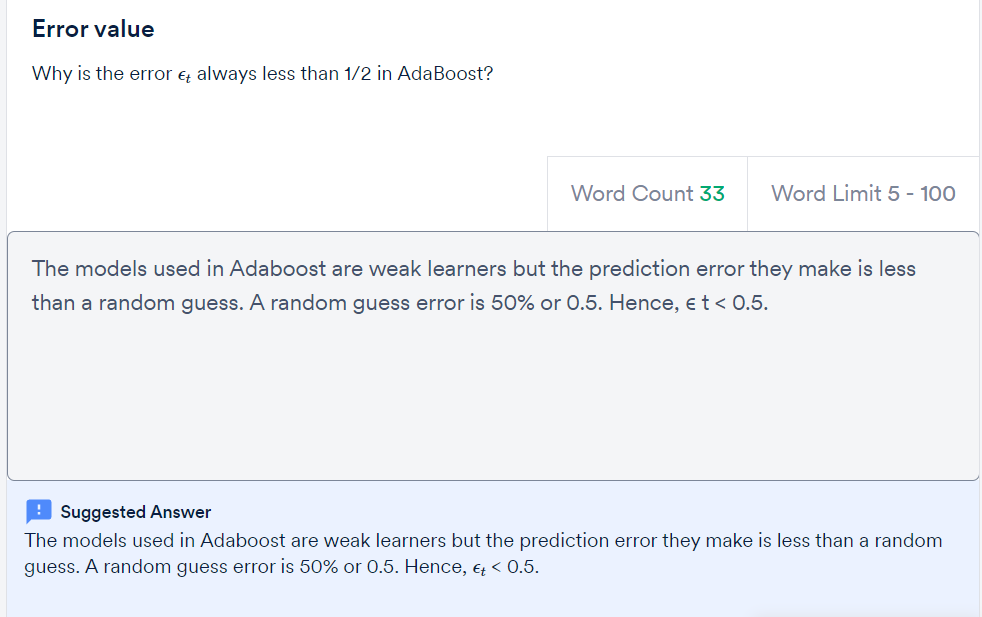


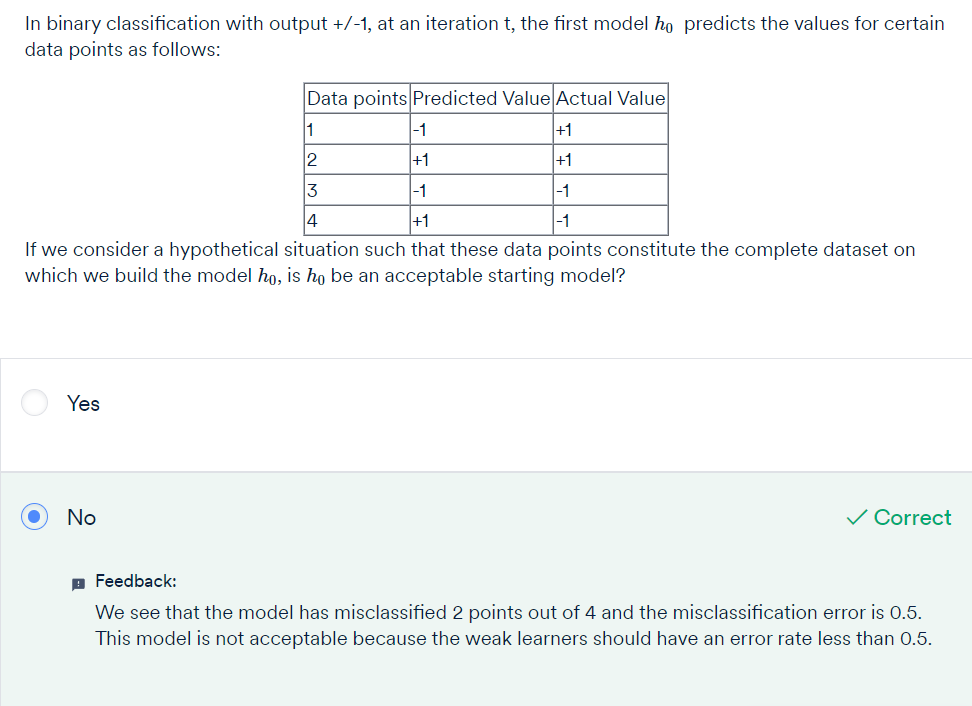


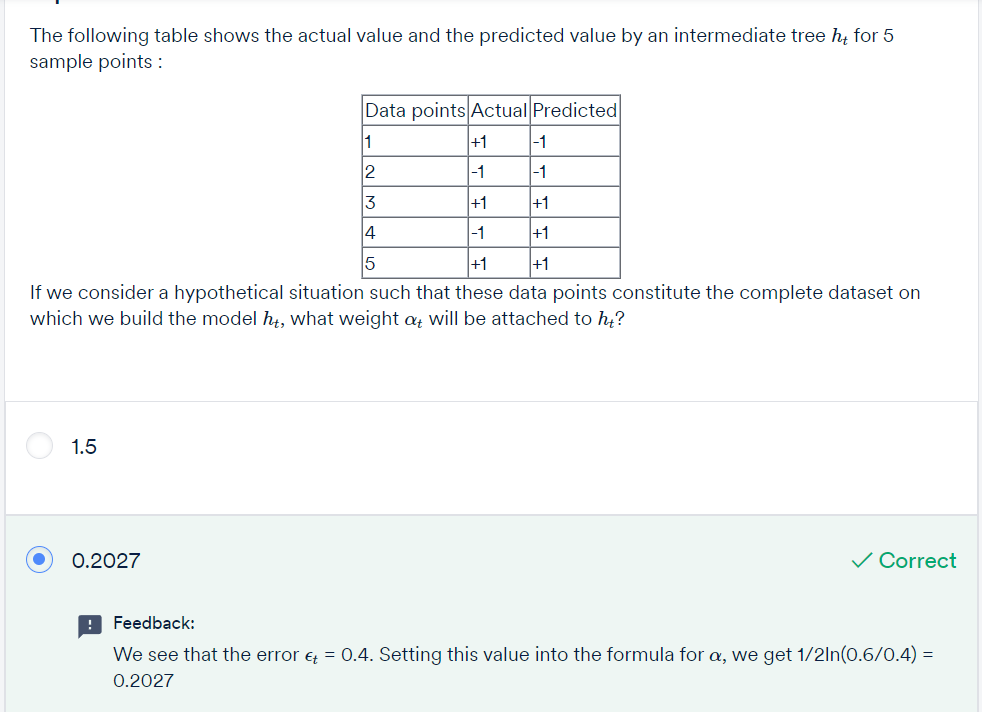


**Practical advice:** Before you apply the AdaBoost algorithm, you should remove the **Outliers**. Since AdaBoost tends to boost up the probabilities of misclassified points and there is a high chance that outliers will be misclassified, it will keep increasing the probability associated with the outliers and make the progress difficult. Some of the ways to identify outliers are:

* Boxplots
* Cook's distance
* Z-score.



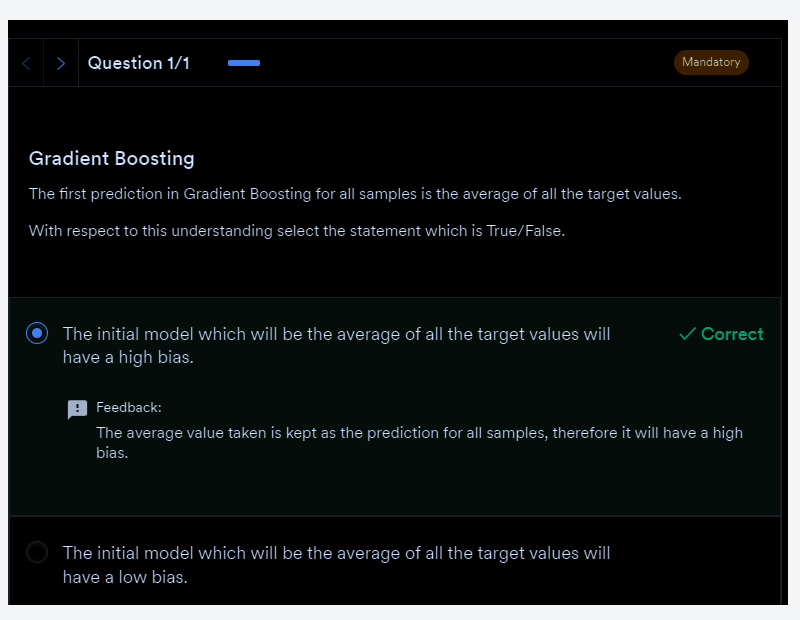


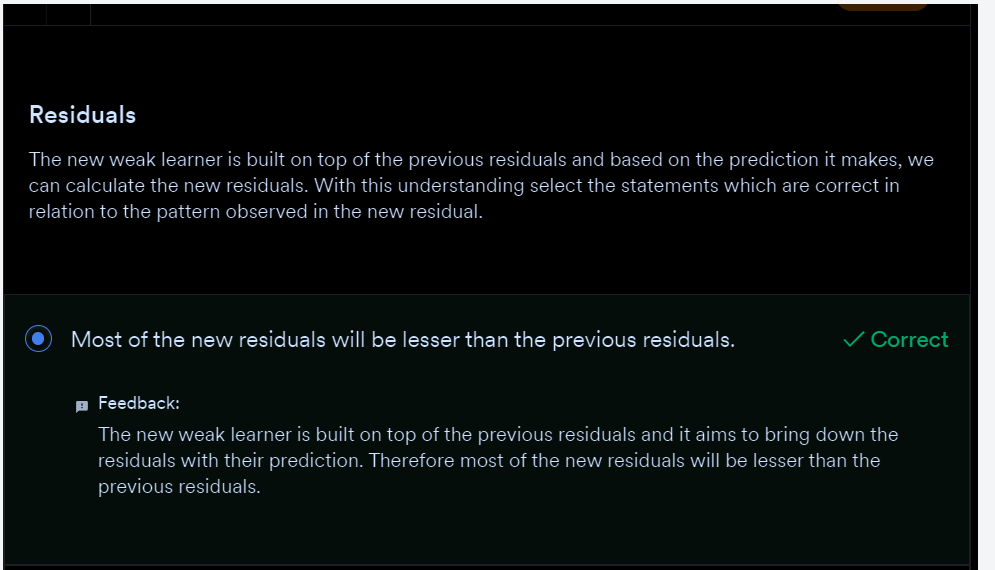


In this session, you learnt the intuition behind boosting and studied the AdaBoost algorithm in detail.

* AdaBoost starts with a uniform distribution of weights over training examples.
* These weights give the importance of the datapoint being considered.
* You will first start with a weak learner h1(x) to create the initial prediction.
* Patterns which are not captured by previous models become the goal for the next model by giving more weightage.
* The next model (weak learner) trains on this resampled data to create the next prediction.
* This process will be repeated till a pre-specified number of trees/models are built.
* In the end, we take a weighted sum of all the weak classifiers to make a strong classifier.

# Gradient Boosting: (GBM)

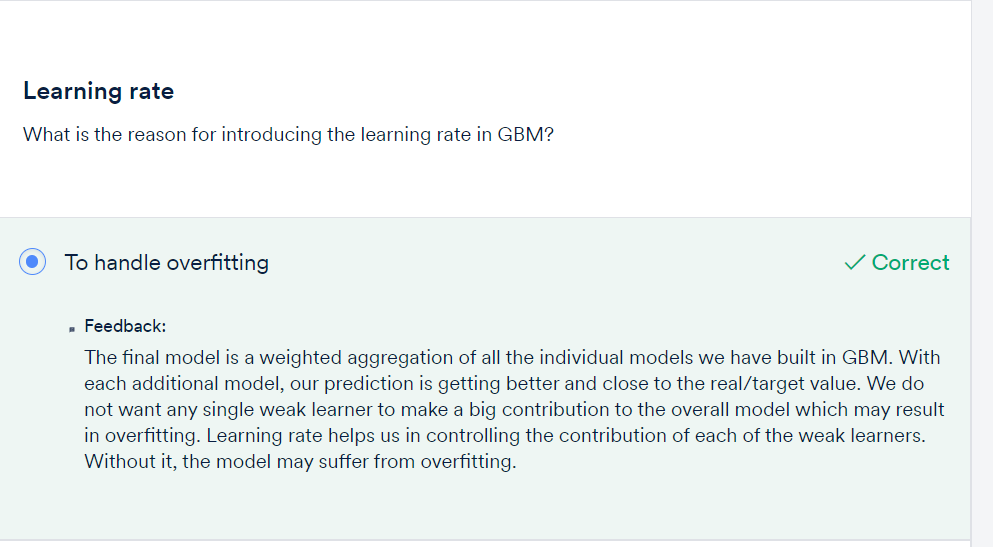


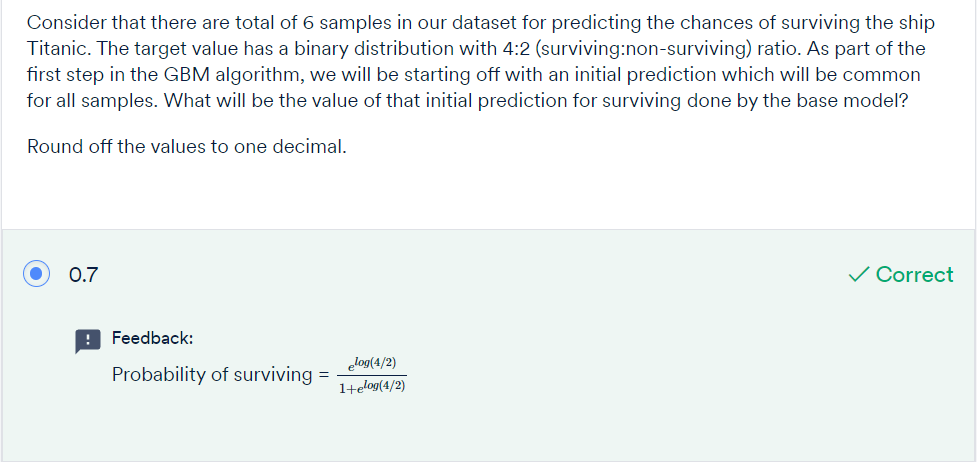


**NOTE**: New prediction = Initial prediction + Learning Rate\*Residuals

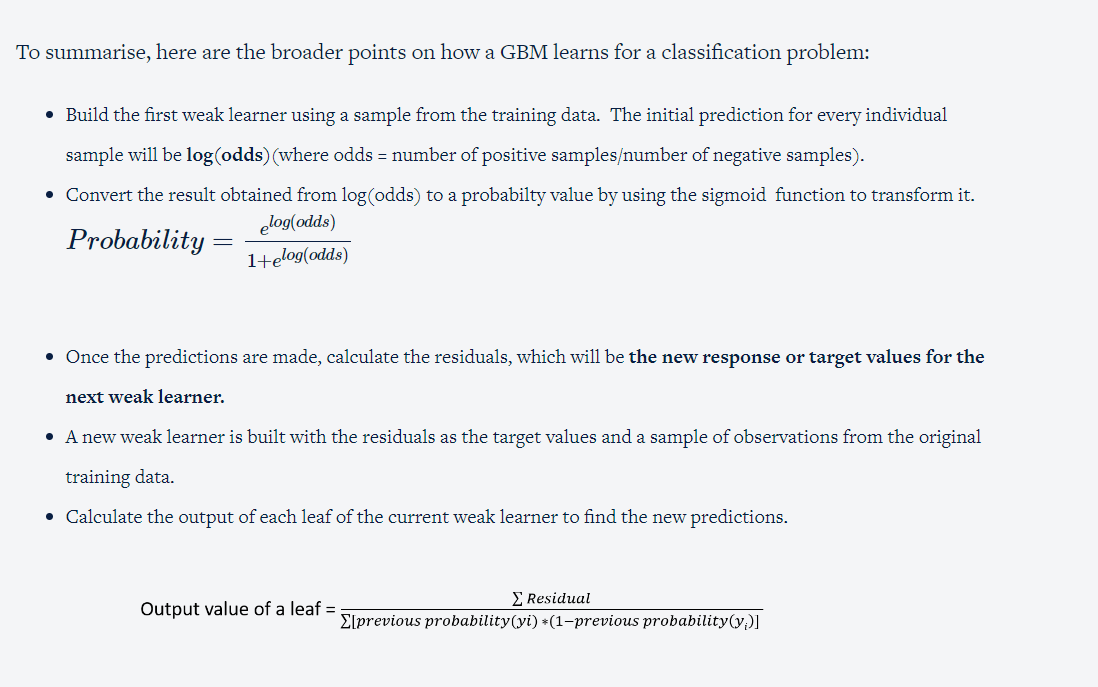
To summarise, here are the broader points on how a GBM learns:

* Build the first weak learner using a sample from the training data; you can consider a decision tree as the weak learner or the base model. It may not necessarily be a stump, can grow a bigger tree but will still be weak, i.e., still not be fully grown.
* Then, predictions are made on the training data using the decision tree which was just built.
* The **negative** **gradient**, in our case the residuals, are computed and these**residuals are the new response or target values for the next weak learner.**
* A new weak learner is built with the residuals as the target values and a sample of observations from the original training data.
* Add the predictions obtained from the current weak learner to the predictions obtained from all the previous weak learners. The predictions obtained at each step are multiplied by the learning rate so that no single model makes a huge contribution to the ensemble thereby avoiding overfitting. Essentially, with the addition of each weak learner, the model takes a very small step in the right direction.
* The next weak learner fits on the residuals obtained till now and these steps are repeated, either for a pre-specified number of weak learners or if the model starts overfitting, i.e., it starts to capture the niche patterns of the training data.
* GBM makes the final prediction by simply adding up the predictions from all the weak learners (multiplied by the learning rate).



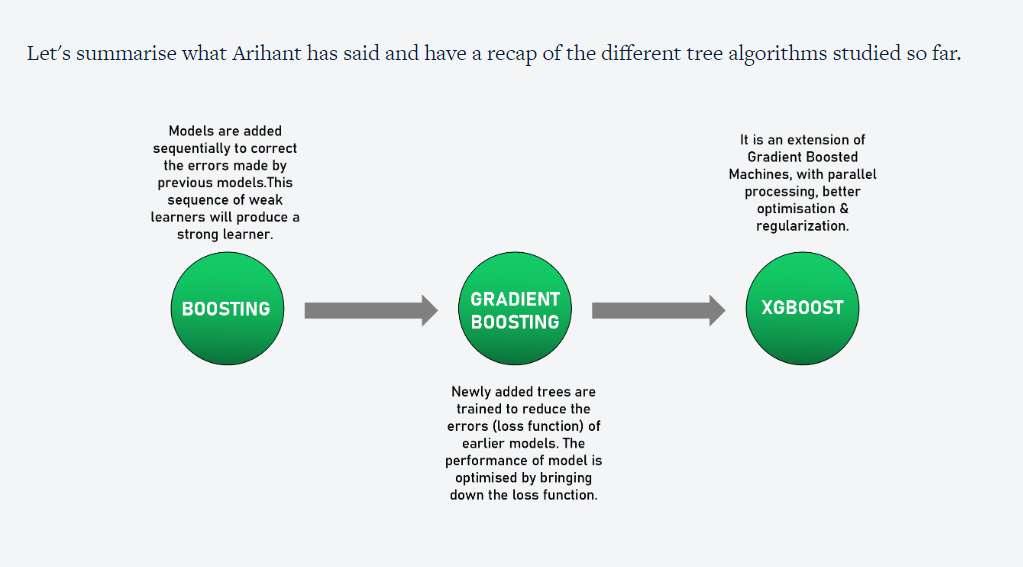






# XGBoost:

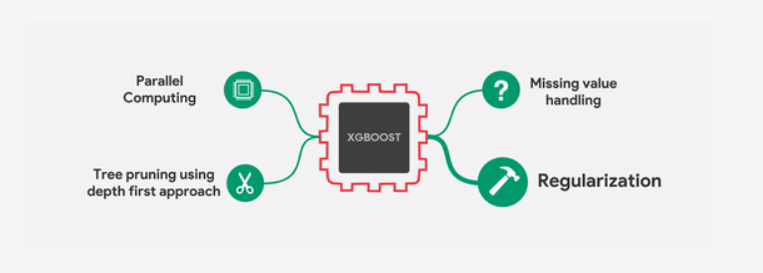
Extreme Gradient Boosting

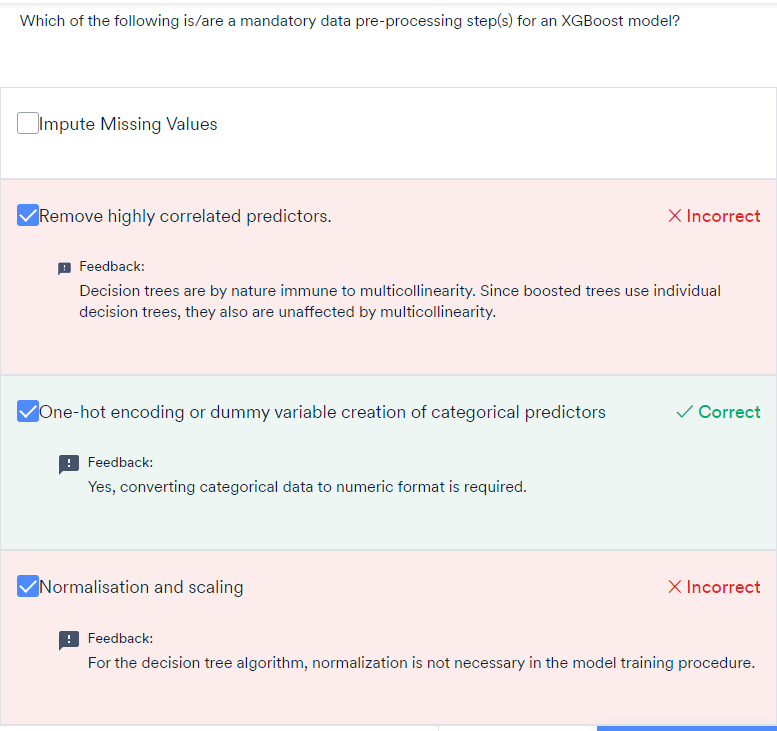


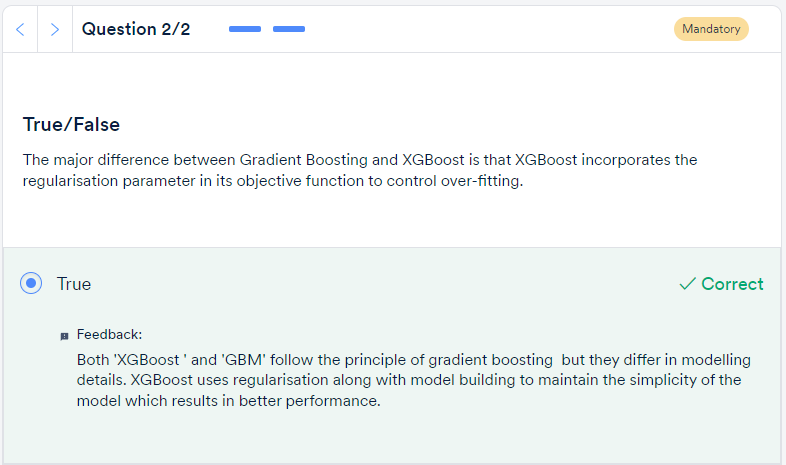
* **AdaBoost** is an iterative way of adding weak learners to form the final model. For this, each model is trained to correct the errors made by the previous one. The sequential model does this by adding more weight to cases with incorrect predictions. Using this approach, the ensemble model will correct itself while learning by focusing on cases/datapoints that are hard to predict correctly.
* Next, let’s discuss**gradient boosting.** You learnt about gradient descent in the previous module. The same principle applies here as well, where the newly added trees are trained to **reduce the errors (loss function)**of earlier models. So, in gradient boosting, you can optimise the performance of the boosted model by bringing down the loss one small step at a time.
* **XGBoost** is an extended version of**gradient boosting**, which uses more accurate approximations to tune the model and find the best fit.

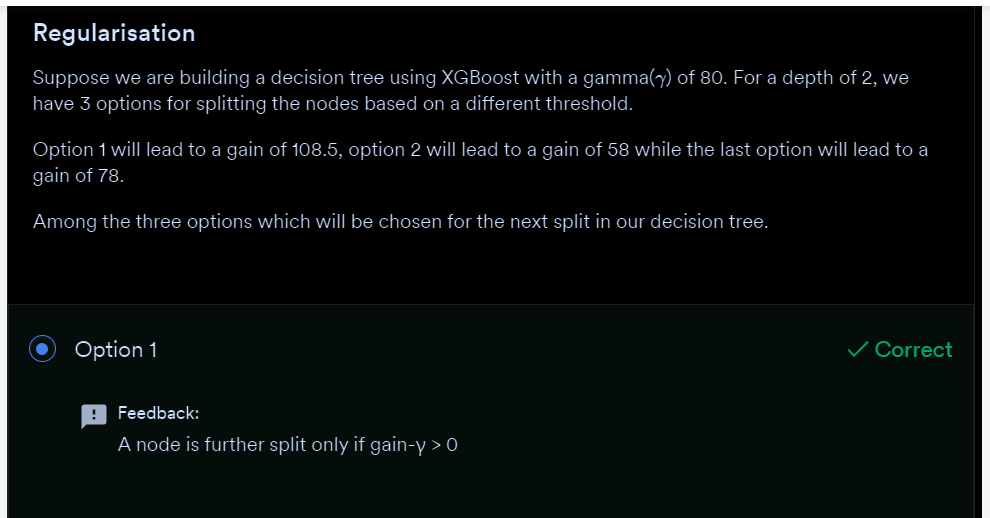
## **Why is XGBoost so good?**

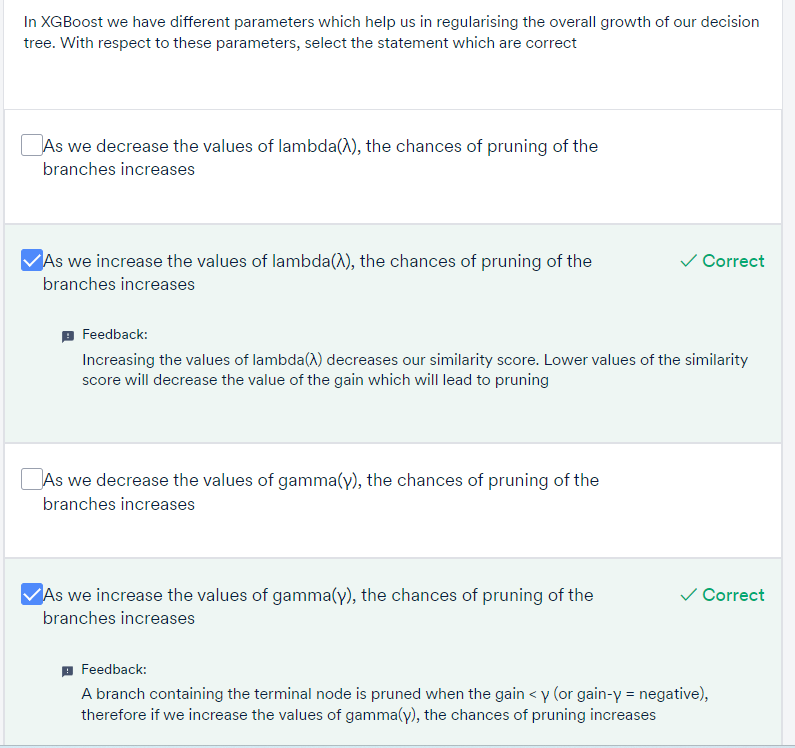
1. **Parallel Computing:** When you run XGBoost, by default it would use all the cores of your laptop/machine enabling its capacity to do parallel computation.
2. **Tree pruning using depth first approach:**XGBoost uses ‘max\_depth’ parameter as specified instead of criterion first, and starts pruning trees backward.
3. **Missing Values:** XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that any trend in missing values (if it exists)  is captured by the model.
4. **Regularization:** The biggest advantage of XGBoost is that it uses regularisation in its objective function which helps to controls the overfitting and simplicity of the model,  leading to better performance.







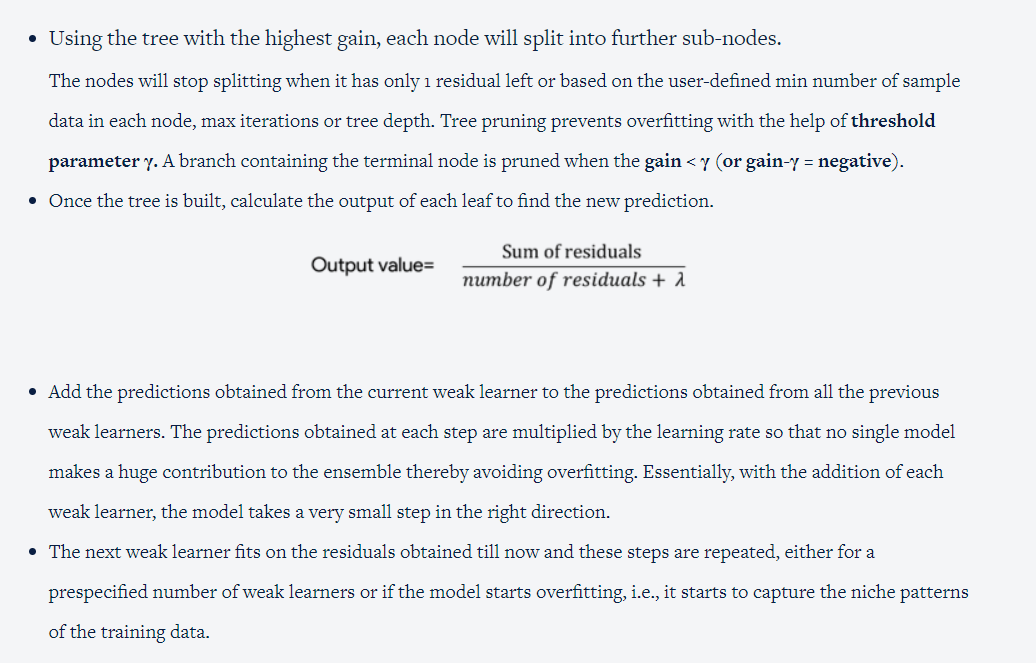




To summarise, here are the broader points on how an XGBoost learns:

* Build the first weak learner which performs the initial prediction on the given dataset. The initial prediction will be 0.5 for both regression & classification tasks.
* The residuals are computed and they will be the new response or target values for the next weak learner.
* A new weak learner is built with the residuals as the target values and a sample of observations from the original training data.
* The new weak learner is created by calculating the **similarity score and gain** for all the constructed trees. **The final tree is the one which has the optimal split i.e highest gain.** The various trees are constructed by splitting the data into two partitions of various possible splits or thresholds. This threshold for root is calculated by taking an average of two close points among the split and the residuals go to the respective leaf.

**Gain = Similarity score(Left leaf) + Similarity score(right leaf) – Similarity score(root node)**



**Hyperparameters - Learning Rate, Number of Trees and Subsampling**

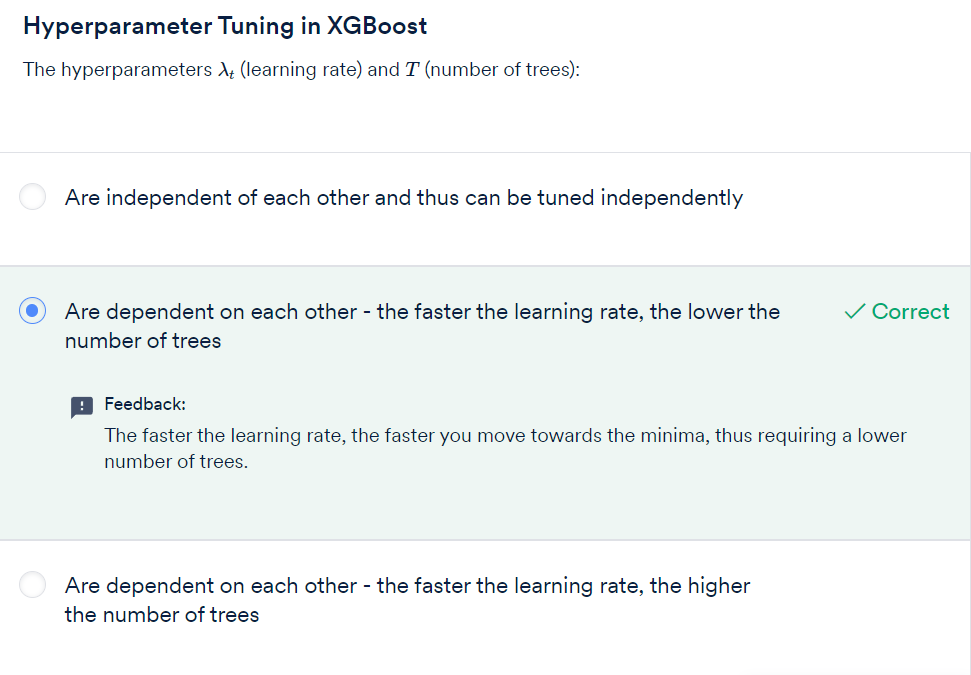
λt, the **learning rate,**is also known as **shrinkage.**It can be used to regularize the gradient tree boosting algorithm. λt typically varies from 0 to 1. Smaller values of λt lead to a larger number of trees T (called n\_estimators in the Python package XGBoost). This is because, with a slower learning rate, you need a larger number of trees to reach the minima.  This, in turn, leads to longer training time. On the other hand, if λt is large, we may reach the same point with a lesser number of trees (n\_estimators), but there is the risk of actually missing the minima altogether (i.e., cross over it) because of the long stride taken at each iteration.

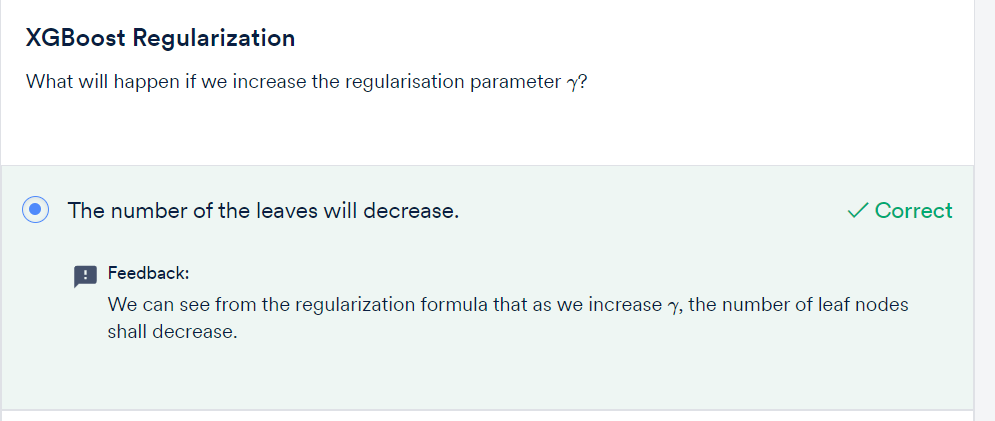
Some other ways of regularisation are explicitly specifying the**number of trees T** and doing **subsampling**. Note that you should not tune both λt and number of trees T together since a high λt implies a low value of T and vice-versa.

**Subsampling** is training the model in each iteration on a fraction of data (similar to how random forests build each tree).  A typical value of subsampling is 0.5 while it ranges from 0 to 1. In random forests, subsampling is critical to ensure diversity among the trees, since otherwise, all the trees will start with the same training data and therefore look similar. This is not a big problem in boosting since each tree is any way built on the residual and gets a significantly different objective function than the previous one.

γ ,**Gamma**is a parameter used to control the pruning of the tree. A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split and makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.

Apart from the previously mentioned hyperparameters, there are other parameters of decision trees like the depth of the tree, the minimum number of samples required for split, etc.





GRADED QUESTIONS:

