4 Boosting Algorithms You Should Know – GBM, XGBoost, LightGBM & CatBoost

MoinDalvs Update README.md 3eb5090 · 2 years ago

README.md	Update README.md	2 years ago
XGBM_Extreme_Gradient_Bo...	Add files via upload	2 years ago
XGBoost_Hyperparameter_Tu...	Add files via upload	2 years ago
diabetes.csv	Add files via upload	2 years ago

README

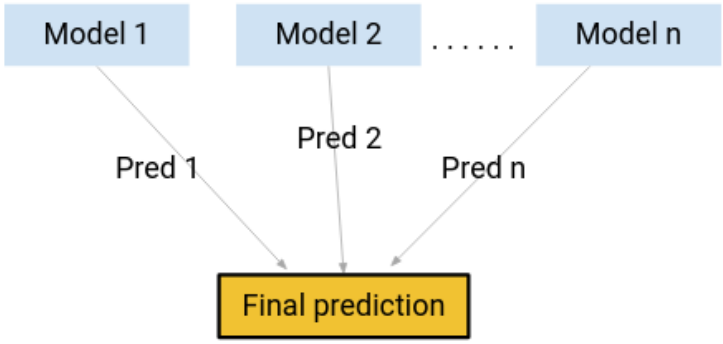
0.1 Table of Contents

1) Quick Introduction to Boosting (What is Boosting?)

Picture this scenario:

You've built a linear regression model that gives you a decent 77% accuracy on the validation dataset. Next, you decide to expand your portfolio by building a k-Nearest Neighbour (KNN) model and a decision tree model on the same dataset. These models gave you an accuracy of 62% and 89% on the validation set respectively.

It's obvious that all three models work in completely different ways. For instance, the linear regression model tries to capture linear relationships in the data while the decision tree model attempts to capture the non-



linearity in the data.
How about, instead of using any one of these models for making the final predictions, we use a combination of all of these models?

I'm thinking of an average of the predictions from these models. By doing this, we would be able to capture more information from the data, right?

That's primarily the idea behind ensemble learning. And where does boosting come in?

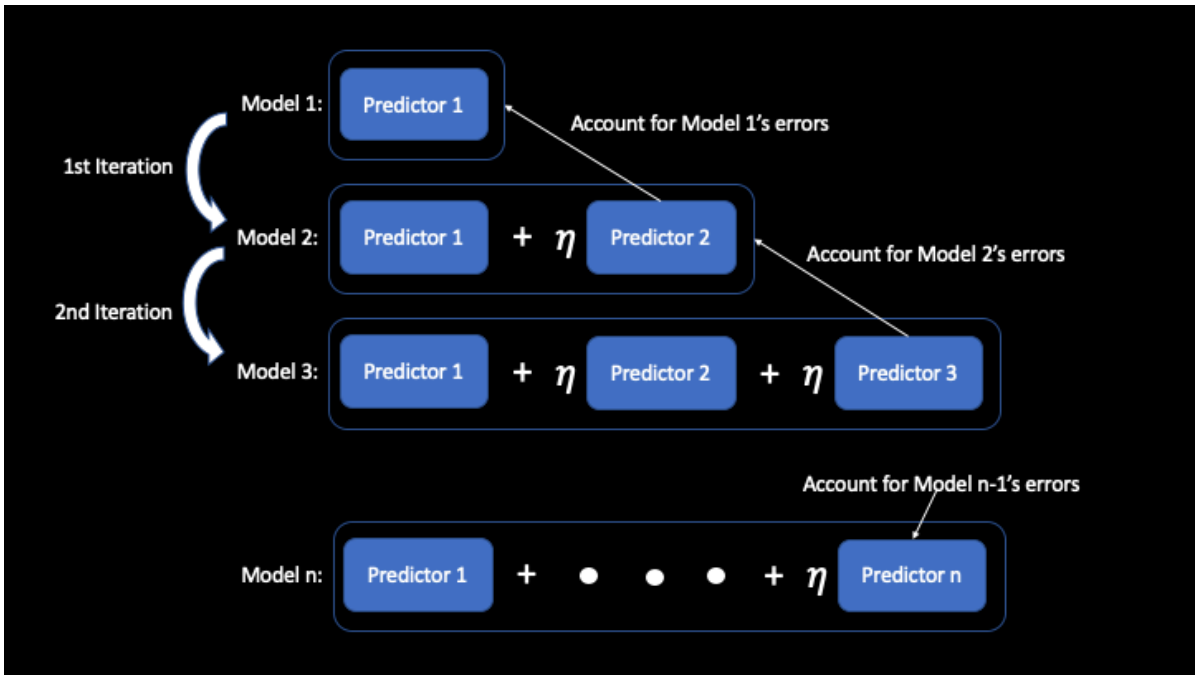
Boosting is one of the techniques that uses the concept of ensemble learning. A boosting algorithm combines multiple simple models (also known as weak learners or base estimators) to generate the final output.

An Ensemble of Weak Learners

When training a new error-predicting model to predict a model's current errors, we regularize its complexity to prevent overfitting. This regularized model will have 'errors' when predicting the original model's 'errors'. It might not necessarily predict 2. Since the new improved model's prediction depends on the new error-predicting model's prediction, it too will have errors albeit lower than before.

To mitigate this, we perform 2 measures. First, we reduce our reliance or trust on any single error predictor by applying a small weight, η (typically between 0 to 0.1) to its output. Then, instead of stopping after 1 iteration of improvement, we repeat the process multiple times, learning new error predictors for newly formed improved models until the accuracy or error is satisfactory. This is summed up using the equations below where x is an input.

- improved_model(x) = current_model(x) + η × error_prediction_model(x)
- current_model(x) = improved_model(x) Repeat above 2 steps till satisfactory
  Typically, the error-predicting model predicts the negative gradient and so, we use addition instead of a subtraction. After every iteration, a new predictor accounting for the errors of the previous model will be learned and added into the ensemble. The number of iterations to perform and η are hyperparameters.



1.1 Gradient Boosting Machine (GBM)

Table of Content

A Gradient Boosting Machine or GBM is an ensemble machine learning algorithm that can be used for classification or regression predictive modeling problems, which combines the predictions from multiple decision trees to generate the final predictions. Keep in mind that all the weak learners in a gradient boosting machine are decision trees. The main objective of Gradient Boost is to minimize the loss function by adding weak learners using a gradient descent optimization algorithm. The generalization allowed arbitrary differentiable loss functions to be used, expanding the technique beyond binary classification problems to support regression, multi-class classification, and more.

The models that form the ensemble, also known as base learners, could be either from the same learning algorithm or different learning algorithms. Bagging and boosting are two widely used ensemble learners. Though these two techniques can be used with several statistical models, the most predominant usage has been with decision trees.

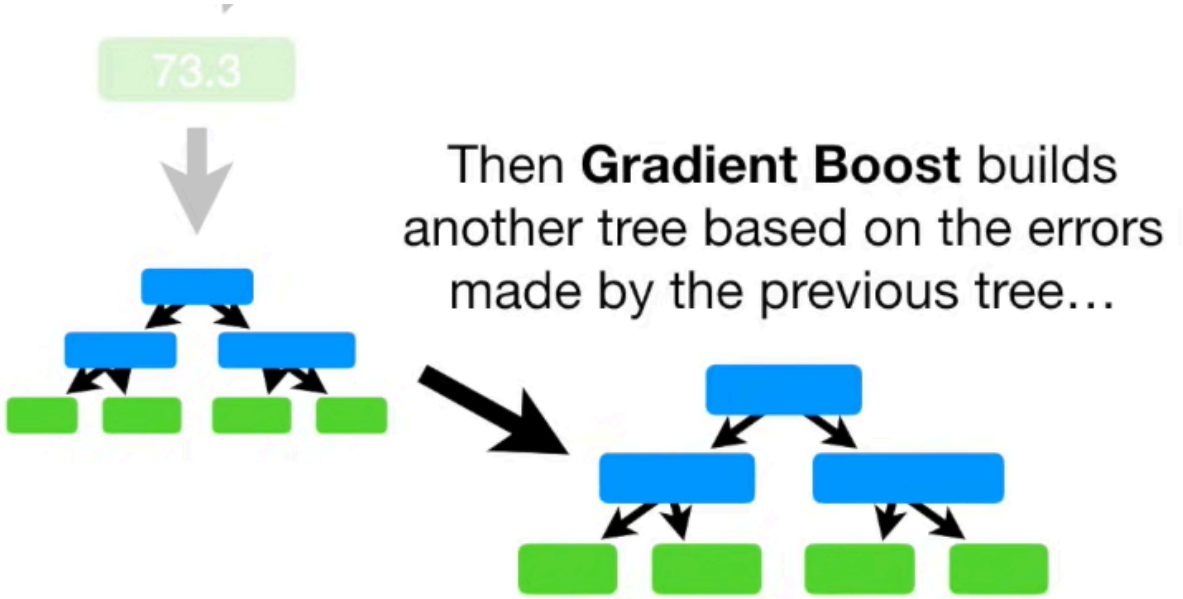
But if we are using the same algorithm, then how is using a hundred decision trees better than using a single decision tree? How do different decision trees capture different signals/information from the data?

Bagging

While decision trees are one of the most easily interpretable models, they exhibit highly variable behavior. Consider a single training dataset that we randomly split into two parts. Now, let's use each part to train a decision tree in order to obtain two models.

When we fit both these models, they would yield different results. Decision trees are said to be associated with high variance due to this behavior. Bagging or boosting aggregation helps to reduce the variance in any learner. Several decision trees which are generated in parallel, form the base learners of bagging technique. Data sampled with replacement is fed to these learners for training. The final prediction is the averaged output from all the learners.

Here is the trick – the nodes in every decision tree take a different subset of features for selecting the best split. This means that the individual trees aren't all the same and hence they are able to capture different signals from the data.

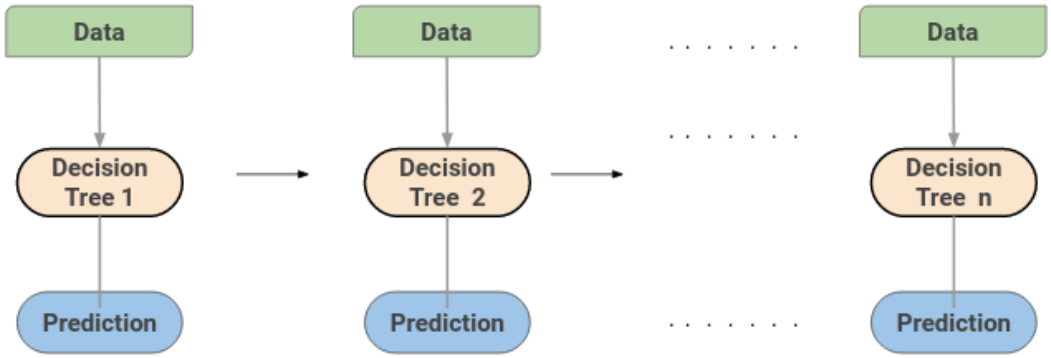


Boosting

Additionaly in boosting, the trees are built sequentially such that each subsequent tree aims to reduce the errors of the previous tree. Each tree learns from its predecessors and updates the residual errors. Hence, the tree that grows next in the sequence will learn from an updated version of the residuals.

The base learners in boosting are weak learners in which the bias is high, and the predictive power is just a tad better than random guessing. Each of these weak learners contributes some vital information for prediction, enabling the boosting technique to produce a strong learner by effectively combining these weak learners. As we already know that errors play a major role in any machine learning algorithm. There are mainly two types of error, bias error and variance error. The final strong learner helps us minimize bring down both the bias and the variance.

In contrast to bagging techniques like Random Forest, in which trees are grown to their maximum extent, boosting makes use of trees with fewer splits. Such small trees, which are not very deep, are highly interpretable. 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.

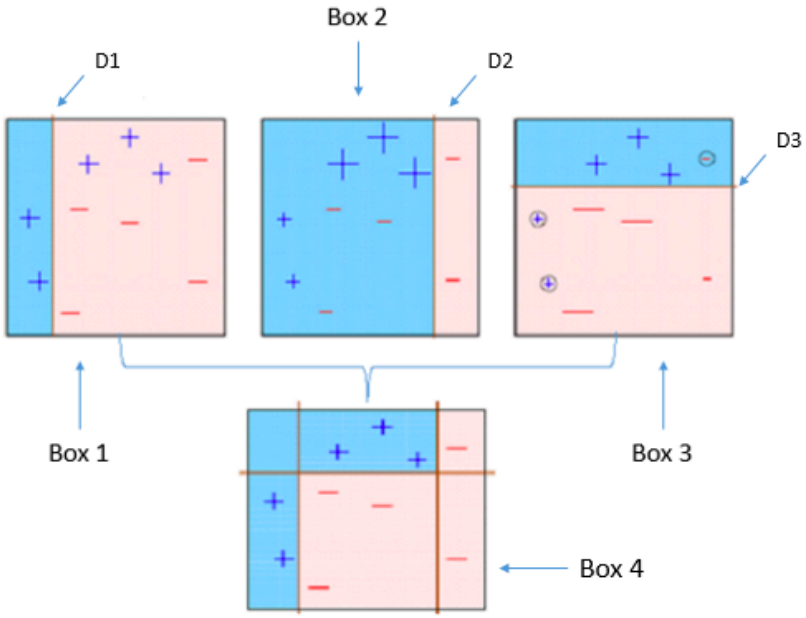


\

The first realization of boosting that saw great success in application was Adaptive Boosting or AdaBoost for short.

AdaBoost Algorithm which is again a boosting method. The weak learners in AdaBoost are decision trees with a single split, called decision stumps for their shortness. This algorithm starts by building a decision stump and then assigning equal weights to all the data points. Then it increases the weights for all the points which are misclassified and lowers the weight for those that are easy to classify or are correctly classified. A new decision stump is made for these weighted data points. The idea behind this is to improve the predictions made by the first stump. New weak learners are added sequentially that focus their training on the more difficult patterns.The main difference between these two algorithms is that Gradient boosting has a fixed base estimator i.e., Decision Trees whereas in AdaBoost we can change the base estimator according to our needs.

AdaBoost uses multiple iterations to generate a single composite strong learner. It creates a strong learner by iteratively adding weak learners. During each phase of training, a new weak learner is added to the ensemble, and a weighting vector is adjusted to focus on examples that were misclassified in previous rounds. The result is a classifier that has higher accuracy than the weak learner classifiers.



Gradient Boosting trains many models in a gradual, additive and sequential manner. The major difference between AdaBoost and Gradient Boosting Algorithm is how the two algorithms identify the shortcomings of weak learners (eg. decision trees). Thus, like AdaBoost, Gradient Boost builds fixed sized trees based on the previous tree's errors, but unlike AdaBoost, each tree can be larger than a stump.In Contrast, Gradient Boost starts by making a single leaf, instead of a tree or a stump. While the AdaBoost model identifies the shortcomings by using high weight data points, gradient boosting performs the same by using gradients in the loss function

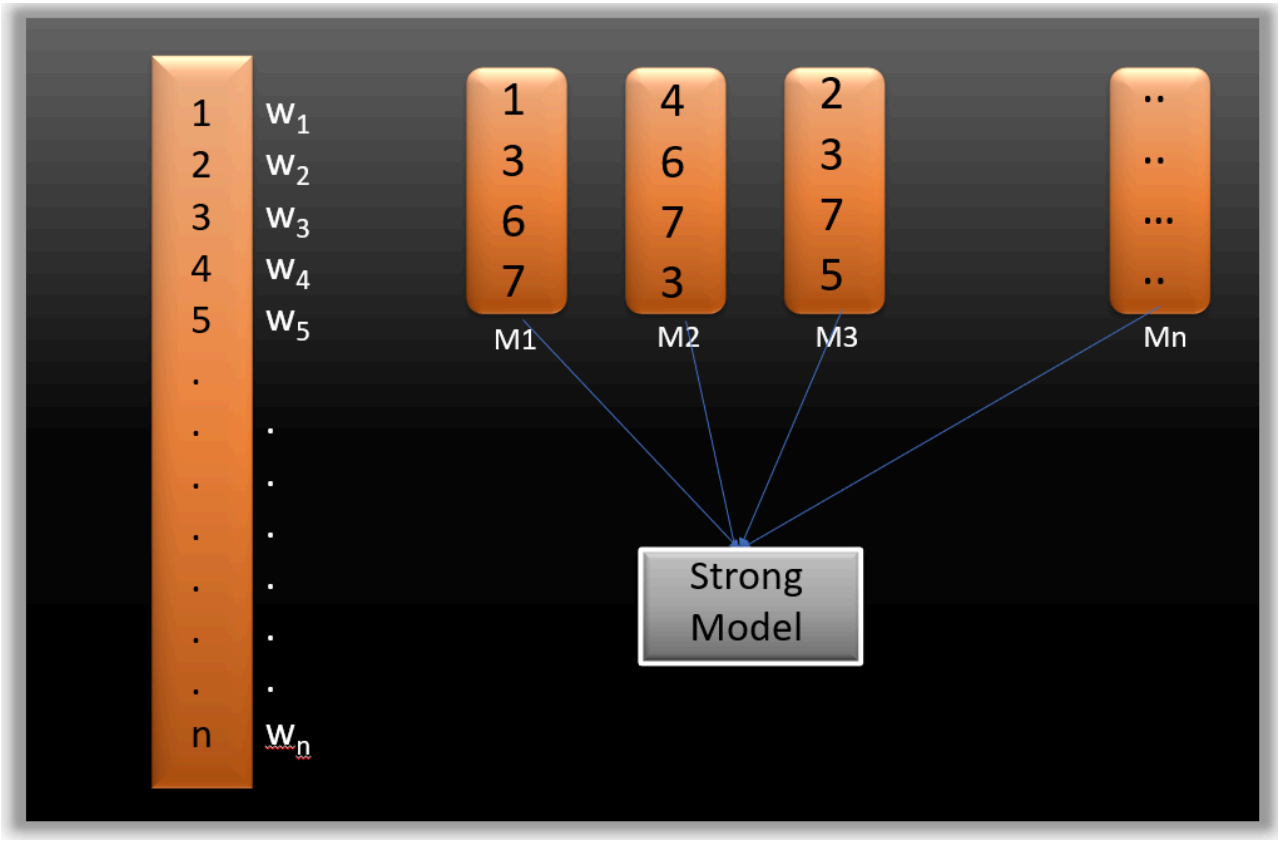
1.2 What is boosting?

While studying machine learning you must have come across this term called Boosting. Boosting is an ensemble learning technique to build a strong classifier from several weak classifiers in series. Boosting algorithms play a crucial role in dealing with bias-variance trade-offs. Unlike bagging algorithms, which only control for high variance in a model, boosting controls both the aspects (bias & variance) and is considered to be more effective.

Below are the few types of boosting algorithms:

1. AdaBoost (Adaptive Boosting)
2. Gradient Boosting
3. XGBoost
4. CATBoost
5. Light GBM

The principle behind boosting algorithms is first we built a model on the training dataset, then a second model is built to rectify the errors present in the first model. Let me try to explain to you what exactly does this means and how does this works.



Suppose you have n data points and 2 output classes (0 and 1). You want to create a model to detect the class of the test data. Now what we do is randomly select observations from the training dataset and feed them to model 1 (M1), we also assume that initially, all the observations have an equal weight that means an equal probability of getting selected.

Remember in ensembling techniques the weak learners combine to make a strong model so here M1, M2, M3....Mn all are weak learners.

Since M1 is a weak learner, it will surely misclassify some of the observations. Now before feeding the observations to M2 what we do is update the weights of the observations which are wrongly classified. You can think of it as a bag that initially contains 10 different color balls but after some time some kid takes out his favorite color ball and put 4 red color balls instead inside the bag. Now off-course the probability of selecting a red ball is higher. This same phenomenon happens in Boosting techniques, when an observation is wrongly classified, its weight get's updated and for those which are correctly classified, their weights get decreased. The probability of selecting a wrongly classified observation gets increased hence in the next model only those observations get selected which were misclassified in model 1.

Similarly, it happens with M2, the wrongly classified weights are again updated and then fed to M3. This procedure is continued until and unless the errors are minimized, and the dataset is predicted correctly. Now when the new datapoint comes in (Test data) it passes through all the models (weak learners) and the class which gets the highest vote is the output for our test data.

1.3 Improvements to Basic Gradient Boosting

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly.

It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting.

In this this section we will look at 4 enhancements to basic gradient boosting:

- Tree Constraints
- Shrinkage
- Random sampling
- Penalized Learning

1. 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.

2. 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.

3. 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.

4. Penalized Gradient Boosting Additional constraints can be imposed on the parameterized trees in addition to their structure.

- L1 regularization of weights.
- L2 regularization of weights.

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. So regularization methods are used to improve the performance of the algorithm by reducing overfitting.

- **Subsampling:** This is the simplest form of regularization method introduced for GBM's. This improves the generalization properties of the model and reduces the computation efforts. Subsampling introduces randomness into the fitting procedure. At each learning iteration, only a random part of the training data is used to fit a consecutive base-learner. The training data is sampled without replacement.
- **Shrinkage:** Shrinkage is commonly used in ridge regression where it shrinks regression coefficients to zero and, thus, reduces the impact of potentially unstable regression coefficients. In GBM's, shrinkage is used for reducing the impact of each additionally fitted base-learner. It reduces the size of incremental steps and thus penalizes the importance of each consecutive iteration. The intuition behind this technique is that it is better to improve a model by taking many small steps than by taking fewer large steps. If one of the boosting iterations turns out to be erroneous, its negative impact can be corrected easily in subsequent steps.
- **Early Stopping:** One important practical consideration that can be derived from Decision Tree is early stopping or tree pruning. This means that if the ensemble was trimmed by the number of trees, corresponding to the validation set minima on the error curve, the overfitting would be circumvented at the minimal accuracy expense. Another observation is that the optimal number of boosts, at which the early stopping is considered, varies concerning the shrinkage parameter λ. Therefore, a trade-off between the number of boosts and λ should be considered.

1.4 Summary:

Gradient boosting involves three elements:

- i. A loss function to be optimized.
- ii. A weak learner to make predictions.
- iii. An additive model to add weak learners to minimize the loss function.

1. 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.

2. 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.

3. Additive Model Trees are added one at a time, and existing trees in the model are not changed.

A gradient descent procedure is used to minimize the loss when adding trees.

Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.

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

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.

Table of Content

Maths Intuition

1.5 Understand Gradient Boosting Algorithm with example (Regression)

Let's understand the intuition behind Gradient boosting with the help of an example. Here our target column is continuous hence we will use Gradient Boosting Regressor.

Following is a sample from a random dataset where we have to predict the car price based on various features. The target column is price and other features are independent features.

Row No.	Cylinder Number	Car Height	Engine Location	Price
1	Four	48.8	Front	12000
2	Six	48.8	Back	16500
3	Five	52.4	Back	15500
4	Four	54.3	Front	14000

\

Step -1 The first step in gradient boosting is to build a base model to predict the observations in the training dataset. For simplicity we take an average of the target column and assume that to be the predicted value as shown below:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

Looking at this may give you a headache, but don't worry we will try to understand what is written here.

Here L is our loss function

Gamma is our predicted value

argmin means we have to find a predicted value/gamma for which the loss function is minimum.

Since the target column is continuous our loss function will be:

$$L = \frac{1}{n}\sum_{i=0}^{n}\left(y_i - \gamma_i\right)^2$$

\

loss function | Gradient Boosting Algorithm Here yi is the observed value

And gamma is the predicted value

Now we need to find a minimum value of gamma such that this loss function is minimum. We all have studied how to find minima and maxima in our 12th grade. Did we use to differentiate this loss function and then put it equal to 0 right? Yes, we will do the same here.

$$\frac{\mathrm{dL}}{\mathrm{d}\gamma} = \frac{2}{2}\left(\sum_{i=0}^{n}\left(y_i - \gamma_i\right)\right) = -\sum_{i=0}^{n}\left(y_i - \gamma_i\right)$$

differentiate loss function Let's see how to do this with the help of our example. Remember that y_i is our observed value and gamma_i is our predicted value, by plugging the values in the above formula we get:

$$L = \frac{1}{2}(12000-\gamma)^2 + \frac{1}{2}(16500-\gamma)^2 + \frac{1}{2}(15500-\gamma)^2 + \frac{1}{2}(14000-\gamma)^2$$

$$\frac{d\mathrm{L}}{d\gamma} = \frac{2}{2}(12000-\gamma)(-1) + \frac{2}{2}(16500-\gamma)(-1) + \frac{2}{2}(15500-\gamma)(-1) + \frac{2}{2}(14000-\gamma)(-1)$$

$$Now\ \frac{d\mathrm{L}}{d\gamma} = 0\ and\ taking\ (-)\ common$$

$$\Rightarrow\ -[12000-\gamma+16500-\gamma+15500-\gamma+14000-\gamma] = 0$$

$$\Rightarrow\ [58000\ -4\gamma] = 0$$

$$\Rightarrow\ 58000 = 4\gamma$$

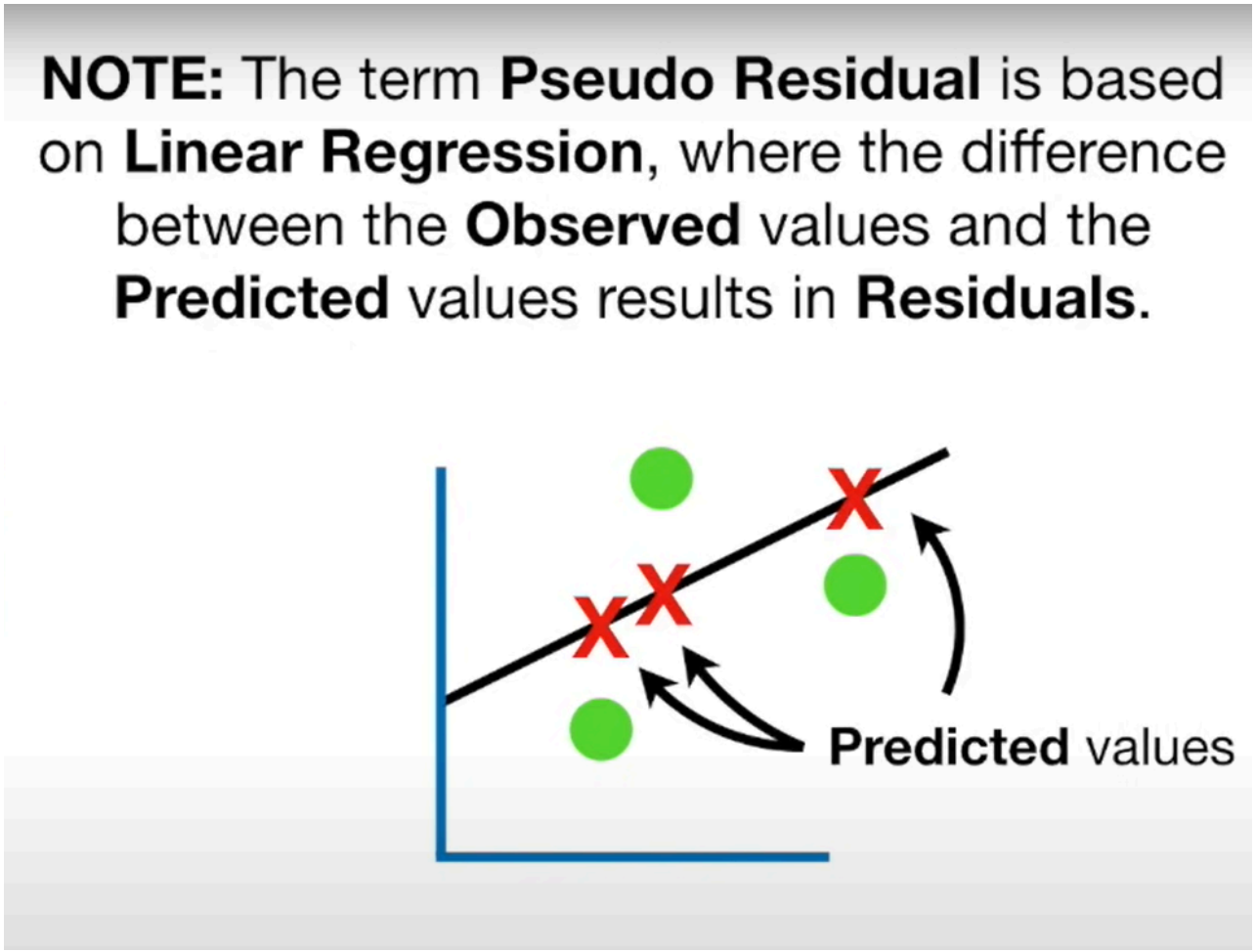
$$\Rightarrow\ \gamma = \frac{58000}{4} = 14500$$

plug values | Gradient Boosting Algorithm We end up over an average of the observed car price and this is why I asked you to take the average of the target column and assume it to be your first prediction.

Hence for gamma=14500, the loss function will be minimum so this value will become our prediction for the base model.

Row No.	Cylinder Number	Car Height	Engine Location	Price	Prediction 1
1	Four	48.8	Front	12000	14500
2	Six	48.8	Back	16500	14500
3	Five	52.4	Back	15500	14500
4	Four	54.3	Front	14000	14500

Step-2 The next step is to calculate the pseudo residuals which are (observed value – predicted value)



Again the question comes why only observed – predicted? Everything is mathematically proved, let's from where did this formula come from. This step can be written as:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

Here F(xi) is the previous model and m is the number of DT made.

The predicted value here is the prediction made by the previous model. In our example the prediction made by the previous model (initial base model prediction) is 14500, to calculate the residuals our formula becomes:

$$(Observed - 14500)$$

Row No.	Cylinder Number	Car Height	Engine Location	Price	Prediction 1	Residual 1
1	Four	48.8	Front	12000	14500	-2500
2	Six	48.8	Back	16500	14500	2000
3	Five	52.4	Back	15500	14500	1000
4	Four	54.3	Front	14000	14500	-500

In the next step, we will build a model on these pseudo residuals and make predictions. Why do we do this? Because we want to minimize these residuals and minimizing the residuals will eventually improve our model accuracy and prediction power. So, using the Residual as target and the original feature Cylinder number, cylinder height, and Engine location we will generate new predictions. Note that the predictions, in this case, will be the error values, not the predicted car price values since our target column is an error now.

Let's say hm(x) is our DT made on these residuals.

Step- 3 In this step we find the output values for each leaf of our decision tree. That means there might be a case where 1 leaf gets more than 1 residual, hence we need to find the final output of all the leaves. TO find the output we can simply take the average of all the numbers in a leaf, doesn't matter if there is only 1 number or more than 1.

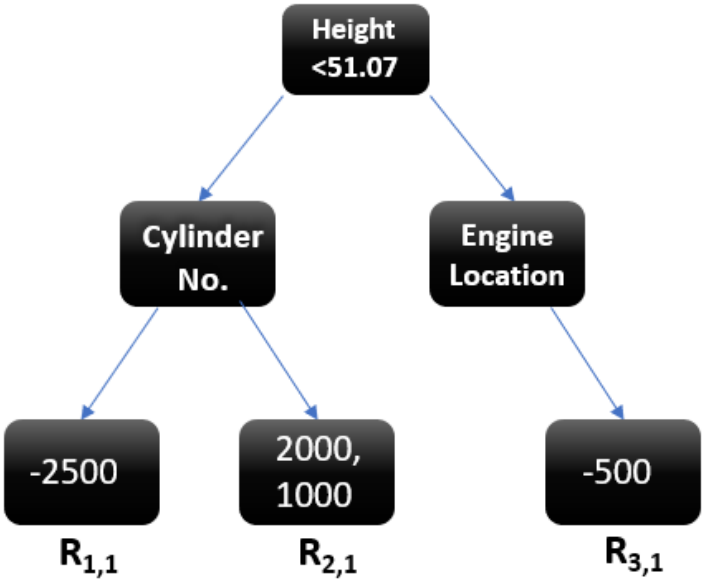
Let's see why do we take the average of all the numbers. Mathematically this step can be represented as:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

Here hm(xi) is the DT made on residuals and m is the number of DT. When m=1 we are talking about the 1st DT and when it is "M" we are talking about the last DT.

The output value for the leaf is the value of gamma that minimizes the Loss function. The left-hand side "Gamma" is the output value of a particular leaf. On the right-hand side [Fm-1(xi)+yhm(xi))] is similar to step 1 but here the difference is that we are taking previous predictions whereas earlier there was no previous prediction.

Let's understand this even better with the help of an example. Suppose this is our regressor tree:



We see 1st residual goes in R1,1 ,2nd and 3rd residuals go in R2,1 and 4th residual goes in R3,1 .

Let's calculate the output for the first leave that is R1,1

$$\gamma_{1,1} = argmin\frac{1}{2}\Big(12000-(14500+\gamma)\Big)^2$$

$$\gamma_{1,1} = argmin\frac{1}{2}\Big(-2500-\gamma\Big)^2$$

Now we need to find the value for gamma for which this function is minimum. So we find the derivative of this equation w.r.t gamma and put it equal to 0.

$$\frac{d}{d\gamma}\frac{1}{2}\Big(-2500-\gamma\Big)^2 = 0$$

$$-2500 - \gamma = 0$$

$$\gamma = -2500$$

Hence the leaf R1,1 has an output value of -2500. Now let's solve for the R2,1

$$\gamma_{2,1} = \text{argmin}\Big[\frac{1}{2}\Big(16500-(14500+\gamma)\Big)^2+\frac{1}{2}\Big(15500-(14500+\gamma)\Big)^2\Big]$$

$$\gamma_{2,1} = \text{argmin}[\frac{1}{2}(2000-\gamma)^2+\frac{1}{2}(1000-\gamma)^2]$$

Let's take the derivative to get the minimum value of gamma for which this function is minimum:

$$\frac{d}{d\gamma}\left[\frac{1}{2}(2000-\gamma)^2+\frac{1}{2}\left(1000-\gamma\right)^2\right] = 0$$

$$2000-\gamma+1000-\gamma=0$$

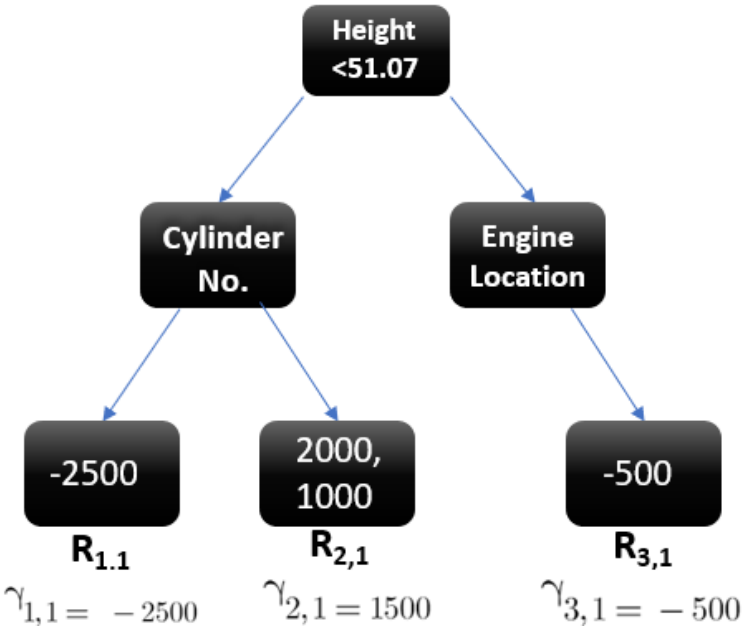
$$3000-2\gamma=0$$

$$\frac{3000}{2}=\gamma$$

$$\gamma=1500$$

We end up with the average of the residuals in the leaf R2,1 . Hence if we get any leaf with more than 1 residual, we can simply find the average of that leaf and that will be our final output.

Now after calculating the output of all the leaves, we get:



Step-4 This is finally the last step where we have to update the predictions of the previous model. It can be updated as:

$$\text{Update the model:}$$

$$F_m(x) = F_{m-1}(x)+\nu_m h_m(x)$$

where m is the number of decision trees made.

Since we have just started building our model so our m=1. Now to make a new DT our new predictions will be:

Here Fm-1(x) is the prediction of the base model (previous prediction) since F1-1=0 , F0 is our base model hence the previous prediction is 14500.

nu is the learning rate that is usually selected between 0-1. It reduces the effect each tree has on the final prediction, and this improves accuracy in the long run. Let's take nu=0.1 in this example.

Hm(x) is the recent DT made on the residuals.

Let's calculate the new prediction now:

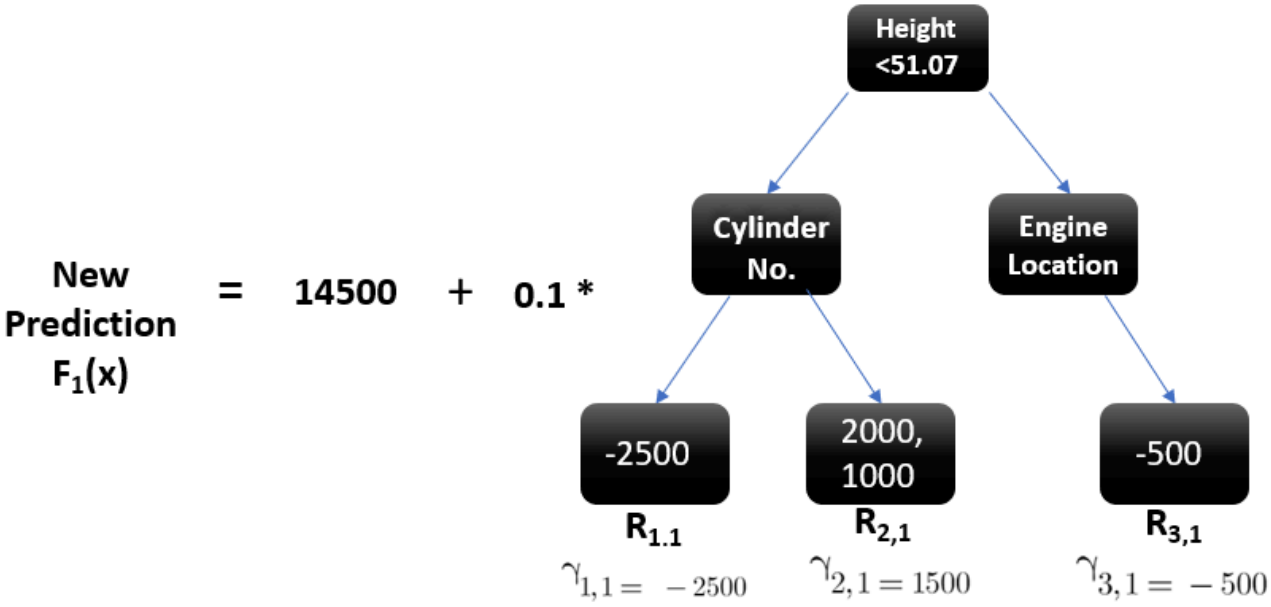


Table of Content

Maths Intuition

1.6 Gradient Boosting Classifier

What is Gradient Boosting Classifier? A gradient boosting classifier is used when the target column is binary. All the steps explained in the Gradient boosting regressor are used here, the only difference is we change the loss function. Earlier we used Mean squared error when the target column was continuous but this time, we will use log-likelihood as our loss function.

Gradient Boosting Algorithm

1. Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

2. for $m = 1$ *to* M:

2-1. Compute residuals $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ $for\ i = 1,\ldots, n$

2-2. Train regression tree with features x against r and create terminal node reasions R_{jm} for $j = 1,\ldots, J_m$

2-3. Compute $\gamma_{jm} = \underset{\gamma}{argmin} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$ $for\ j = 1,\ldots, J_m$

2-4. Update the model:

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$$

Let's see how this loss function works, The first step is creating an initial constant prediction value F_0. L is the loss function and we are using log loss (or more generally called cross-entropy loss) for it.

Step 1

1. Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

$$L = -(y_i \cdot log(p) + (1 - y_i) \cdot log(1 - p))$$

y_i is our classification target and it is either 0 or 1. p is the predicted probability of class 1. You might see L taking different values depending on the target class y_i.

$$L = \begin{cases} -log(p) & if\ y_i = 1 \\ -log(1-p) & if\ y_i = 0 \end{cases}$$

As −log(x) is the decreasing function of x, the better the prediction (i.e. increasing p for y_i=1), the smaller loss we will have.

argmin means we are searching for the value γ (gamma) that minimizes $\Sigma L(y_i,\gamma)$. While it is more straightforward to assume γ is the predicted probability p, we assume γ is log-odds as it makes all the following computations easier. For those who forgot the log-odds definition , it is defined as log(odds) = log(p/(1-p)).

To be able to solve the argmin problem in terms of log-odds, we are transforming the loss function into the function of log-odds.

Our first step in the gradient boosting algorithm was to initialize the model with some constant value, there we used the average of the target column but here we'll use log(odds) to get that constant value. The question comes why log(odds)?

When we differentiate this loss function, we will get a function of log(odds) and then we need to find a value of log(odds) for which the loss function is minimum.

Confused right? Okay let's see how it works:

Let's first transform this loss function so that it is a function of log(odds), I'll tell you later why we did this transformation.

$$
\begin{aligned}
L &= -\left[\sum_{i=1}^{n} y_i \log(p) + \left(1 - y_i\right) log(1-p)\right] \\
&= -y * \log(p) - (1-y) * \log(1-p) \\
&= -y * log(p) - log(1-p) + y * log(1-p) \\
&= y * \left[log(p) - log(1-p)\right] - log(1-p) \\
&= -y * \left[\frac{\log(p)}{\log(1-p)}\right] - log(1-p) \\
&= -y * log\left(\frac{p}{1-p}\right) - log(1-p)
\end{aligned}
$$

Now we might want to replace p in the above equation with something that is expressed in terms of log-odds. By transforming the log-odds expression shown earlier, p can be represented by log-odds:

$$log\left(\frac{p}{1-p}\right) = log(odds)$$

$$\frac{p}{1-p} = e^{log(odds)}$$

$$p = (1-p)e^{log(odds)}$$

$$\left(1 + e^{log(odds)}\right)p = e^{log(odds)}$$

$$p = \frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

Then, we are substituting this value for p in the previous L equation and simplying it.

p is replaced with *this*

$$L = -\left(y_i \cdot log(odds) + log\left(1 - \frac{e^{log(odds)}}{1 + e^{log(odds)}}\right)\right)$$

$$= -\left(y_i \cdot log(odds) + log\left(\frac{1}{1 + e^{log(odds)}}\right)\right)$$

$$= -\left(y_i \cdot log(odds) + \underbrace{log(1)}_{=0} - log\left(1 + e^{log(odds)}\right)\right)$$

$$= -\left(y_i \cdot log(odds) - log\left(1 + e^{log(odds)}\right)\right)$$

Now this is our loss function, and we need to minimize it, for this, we take the derivative of this w.r.t to log(odds) and then put it equal to 0,

$$\frac{\partial}{\partial log(odds)} \sum_{i=1}^{n} L = -\frac{\partial}{\partial log(odds)} \sum_{i=1}^{n} \left[y_i \cdot log(odds) - log\left(1 + e^{log(odds)}\right)\right]$$

$$= -\sum_{i=1}^{n} y_i + n\frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

$$= -\sum_{i=1}^{n} y_i + np$$

Applying chain rule to get derivative of this

$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$

\

In the equations above, we replaced the fraction containing log-odds with p to simplify the equation. Next, we are setting ∂ΣL/∂log(odds) equal to 0 and solving it for p.

$$-\sum_{i=1}^{n} y_i + np = 0$$

$$np = \sum_{i=1}^{n} y_i$$

$$p = \frac{1}{n}\sum_{i=1}^{n} y_i = \bar{y}$$

In this binary classification problem, y is either 0 or 1. So, the mean of y is actually the proportion of class 1. You might now see why we used p = mean(y) for our initial prediction.

As γ is log-odds instead of probability p, we are converting it into log-odds.

$$F_0(x) = \overset{*}{\gamma} = log\left(\frac{\bar{y}}{1-\bar{y}}\right)$$

Step2

> 2. for $m = 1$ *to* M:

The whole step2 processes from 2–1 to 2–4 are iterated M times. M denotes the number of trees we are creating and the small m represents the index of each tree.

Step2-1

> 2-1. Compute residuals $r_{im} = -\left[\frac{\partial L(y_i,\ F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ $for\ i = 1,\ldots, n$

We are calculating residuals $r_{i}m$ by taking a derivative of the loss function with respect to the previous prediction $Fm\text{-}_1$ and multiplying it by −1. As you can see in the subscript index, $r_{i}m$ is computed for each single sample i. Some of you might be wondering why we are calling this $r_{i}m$ residuals. This value is actually negative gradient that gives us the directions (+/−) and the magnitude in which the loss function can be minimized. You will see why we are calling it residuals shortly. By the way, this technique where you use a gradient to minimize the loss on your model is very similar to gradient descent technique which is typically used to optimize neural networks. (In fact, they are slightly different from each other.

Let's compute the residuals here. $Fm\text{-}_1$ in the equation means the prediction from the previous step. In this first iteration, it is F_0. As in the previous step, we are taking a derivative of L with respect to log-odds instead of p since our prediction Fm is log-odds. Below we are using L expressed by log-odds which we got in the previous step.

$$r_{im} = -\frac{\partial}{\partial log(odds)} L$$
$$= \frac{\partial}{\partial log(odds)} \left[y_i \cdot log(odds) - log\left(1 + e^{log(odds)}\right) \right]$$
$$= y_i - \frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

In the previous step, we also got this equation:

$$p = \frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

So, we can replace the second term in r_{im} equation with p.

$$r_{im} = y_i - p$$

You might now see why we call r residuals. This also gives us interesting insight that the negative gradient that provides us the direction and the magnitude to which the loss is minimized is actually just residuals.

----------OR----------

$$\frac{\mathrm{dL}}{\mathrm{d}[\log(\mathrm{odds})]} = -\mathrm{y} + \frac{\mathrm{e}^{\log(\mathrm{odds})}}{1 + \mathrm{e}^{\log(\mathrm{odds})}}$$

$$\text{We know } \frac{\mathrm{e}^{\log(\mathrm{odds})}}{1 + \mathrm{e}^{\log(\mathrm{odds})}} = p, \textit{ hence we can substitue } p$$

$$\frac{\mathrm{dL}}{\mathrm{d}[log(\mathrm{odds})]} = -\mathrm{y} + \mathrm{p}$$

Here y are the observed values
You must be wondering that why did we transform the loss function into the function of log(odds). Actually, sometimes it is easy to use the function of log(odds), and sometimes it's easy to use the function of predicted probability "p".

It is not compulsory to transform the loss function, we did this just to have easy calculations.

Hence the minimum value of this loss function will be our first prediction (base model prediction)

Now in the Gradient boosting regressor our next step was to calculate the pseudo residuals where we multiplied the derivative of the loss function with -1. We will do the same but now the loss function is different, and we are dealing with the probability of an outcome now.

$$\frac{dL}{d[log(odds)]} = -y + p$$

$$\frac{dL}{d[log(odds)]} = -(-y + p) = (y - p) = (observed - predicted)$$

After finding the residuals we can build a decision tree with all independent variables and target variables as "Residuals".

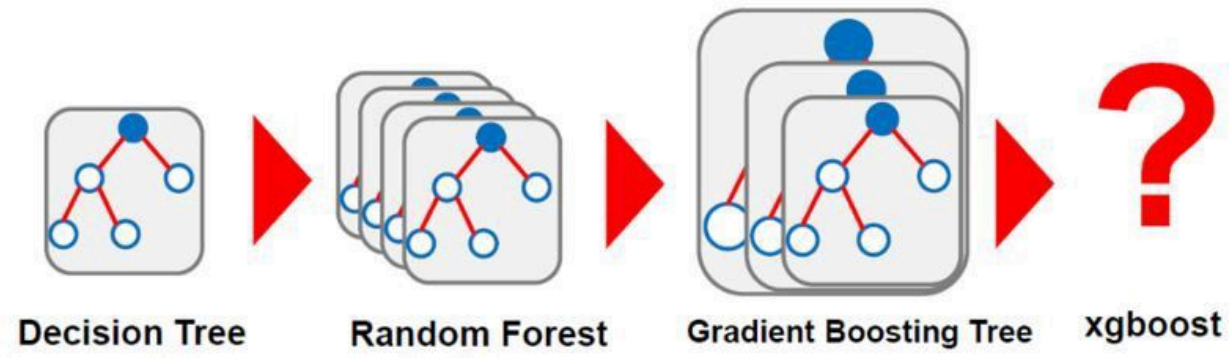
Now when we have our first decision tree, we find the final output of the leaves because there might be a case where a leaf gets more than 1 residuals, so we need to calculate the final output value.

$$\gamma = \frac{\sum_{i=1}^{n} Residual_i}{\sum_{i=1}^{n} [Previous\ probability_i \times (1 - Previous\ probability_i)]}$$

Finally, we are ready to get new predictions by adding our base model with the new tree we made on residuals.

Table of Content

2. Extreme Gradient Boosting Machine (XGBM)



XGBoost is an extension to gradient boosted decision trees (GBM) and specially designed to improve speed and performance. In fact, XGBoost is simply an improvised version of the GBM algorithm! The working procedure of XGBoost is the same as GBM. `Regularized Learning`, `Gradient Tree Boosting` and `Shrinkage and Column Subsampling`. The trees in XGBoost are built sequentially, trying to correct the errors of the previous trees. It is an implementation of Gradient Boosting machines which exploits various optimizations to train powerful predictive models very quickly.

2.1 XGBoost Features

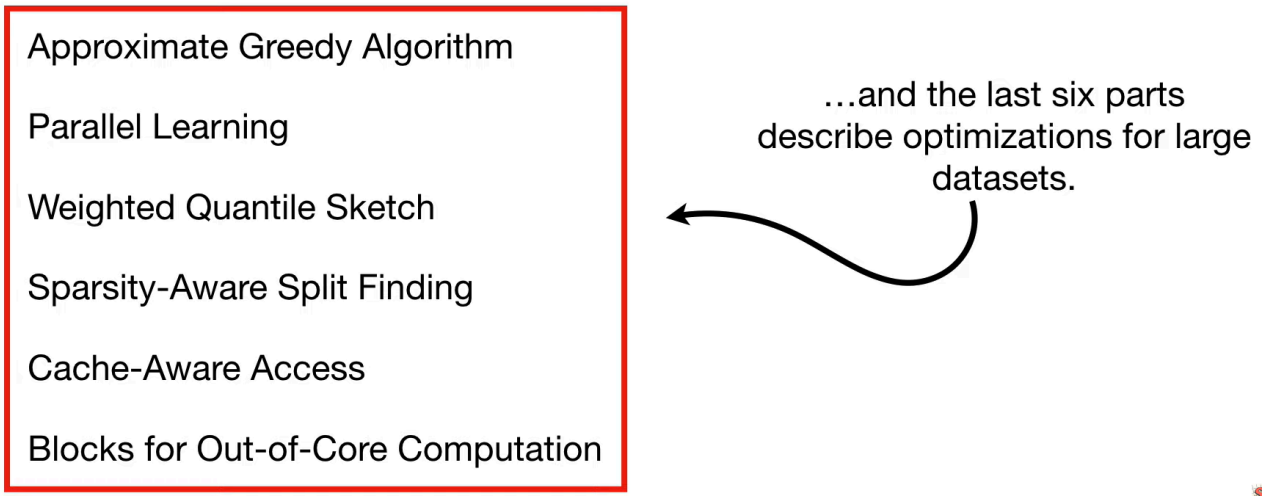
- **Regularized Learning:** The regularization term helps to smooth the final learned weights to avoid over-fitting. The regularized objective will tend to select a model employing simple and predictive functions. Gradient Tree Boosting: The tree ensemble model cannot be optimized using traditional optimization methods in Euclidean space. Instead, the model is trained in an additive manner.
- **Shrinkage and Column Subsampling:** Besides the regularized objective, two additional techniques are used to further prevent overfitting. The first technique is shrinkage introduced by Friedman. Shrinkage scales newly added weights by a factor η after each step of tree boosting. Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each tree and leaves space for future trees to improve the model.
- The second technique is the column (feature) subsampling.
- **Column and Row Subsampling** — To reduce training time, XGBoost provides the option of training every tree with only a randomly sampled subset of the original data rows where the size of this subset is determined by the user. The same applies to the columns/features of the dataset. Apart from savings in training time, subsampling the columns during training has the effect of decorrelating the trees which can reduce overfitting and boost model performance. This technique is used in Random Forest. Column sub-sampling prevents over-fitting even more so than the traditional row sub-sampling. The usage of column sub-samples also speeds up computations of the parallel algorithm.

But there are certain features that make XGBoost slightly better than GBM:

- One of the most important points is that XGBM implements parallel preprocessing (at the node level) which makes it faster than GBM and that means using Parallel learning to split up the dataset so that multiple computers can work on it at the same time.
- XGBoost also includes a variety of regularization techniques that reduce overfitting and improve overall performance. You can select the regularization technique by setting the hyperparameters of the XGBoost algorithm
- Additionally, if you are using the XGBM algorithm, you don't have to worry about imputing missing values in your dataset. The XGBM model can handle the missing values on its own. During the training process, the model learns whether missing values should be in the right or left node.

In other words, the first three parts give us a conceptual idea of How XGBoost is fit to training data and how it makes predictions

and the other parts we are going to discuss are going to describe optimization techniques for large datasets



2.2 XGBM Optimizations:

- **Exact Greedy Algorithm:** The main problem in tree learning is to find the best split. This algorithm enumerates all the possible splits on all the features. It is computationally demanding to enumerate all the possible splits for continuous features.

- **Approximate Algorithm:** The exact greedy algorithm is very powerful since it enumerates overall possible splitting points greedily. However, it is impossible to efficiently do so when the data does not fit entirely into memory. Approximate Algorithm proposes candidate splitting points according to percentiles of feature distribution. The algorithm then maps the continuous features into buckets split by these candidate points, aggregates the statistics, and finds the best solution among proposals based on the aggregated statistics. So when we have huge training dataset, XGBoost uses an Approximate Greedy Algorithm.
- **Weighted Quantile Sketch:** Weighted Quantile Sketch merges the data into an approximate histogram for finding approximate best split — Before finding the best split, we form a histogram for each feature. The boundaries of the histogram bins are then used as candidate points for finding the best split. In the Weighted Quantile Sketch, the data points are assigned weights based on the "confidence" of their current predictions and the histograms are built such that each bin has approximately the same total weight (as opposed to the same number of points in the traditional quantile sketch). As a result, more candidate points and thus, a more detailed search will exist in areas where the model is doing poorly. One important step in the approximate algorithm is to propose candidate split points. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data.
- **Parallelization for faster tree building process** — When finding optimal splits, the trying of candidate points can be parallelized at the feature/column level. For example, core 1 can be finding the best split point and its corresponding loss for feature A while core 2 can be doing the same for feature B. In the end, we compare the losses and use the best one as the split point.
- **Sparsity-aware Split Finding:** In many real-world problems, it is quite common for the input x to be sparse. There are multiple possible causes for sparsity: Presence of missing values in the data Frequent zero entries in the statistics Artifacts of feature engineering such as one-hot encoding The default direction is chosen based on which reduces the Loss more. On top of this, XGBoost ensures that sparse data are not iterated over during the split finding process, preventing unnecessary computation. It is important to make the algorithm aware of the sparsity pattern in the data. XGBoost handles all sparsity patterns in a unified way.
- **Hardware Optimizations** — XGBoost stores the frequently used gs and hs in the cache to minimize data access costs. When disk usage is required (due to data not fitting into memory), the data is compressed before storage, reducing the IO cost involved at the expense of some compression computation. If multiple disks exist, the data can be sharded to increase disk reading throughput.

2.3 System Features

The library provides a system for use in a range of computing environments, not least:

- **Parallelization:** Parallelization of tree construction using all of your CPU cores during training. Collecting statistics for each column can be parallelized, giving us a parallel algorithm for split finding.
- **Cache-aware Access:** XGBoost has been designed to make optimal use of hardware. This is done by allocating internal buffers in each thread, where the gradient statistics can be stored.
- **Distributed Computing** for training very large models using a cluster of machines.
- **Out-of-Core Computing** for very large datasets that don't fit into memory.
- **Cache Optimization** of data structures and algorithm to make the best use of hardware.
- **Column Block for Parallel Learning**: The most time-consuming part of tree learning is to get the data into sorted order. In order to reduce the cost of sorting, the data is stored in the column blocks in sorted order in compressed format.

2.4 Algorithm Features

The implementation of the algorithm was engineered for the efficiency of computing time and memory resources. A design goal was to make the best use of available resources to train the model. Some key algorithm implementation features include:

- **Sparse Aware implementation** with automatic handling of missing data values.
- **Block Structure** to support the parallelization of tree construction.
- **Continued Training** so that you can further boost an already fitted model on new data.

2.5 Weak Learner Tree Splitting

So far, we got the t-th step object function, next step is to build the t-th tree, and this tree should be constructed to reduce object function value as much as possible.

In order to build a tree to reduce object function value, we only allow node split which can reduce object function value, and looking for a best split which can reduce the most.

So in each split we measure the object function value reduce by Tree Object function value(After Node Split) — (Before Node Split)

$$\text{Gain} = \text{Left}_{\text{Similarity}} + \text{Right}_{\text{Similarity}} - \text{Root}_{\text{Similarity}}$$

Gain is how much object function value reduced in the split.

$Left_{Similarity}$ is left splitting child leaf

$Right_{Similarity}$ is right splitting leaf

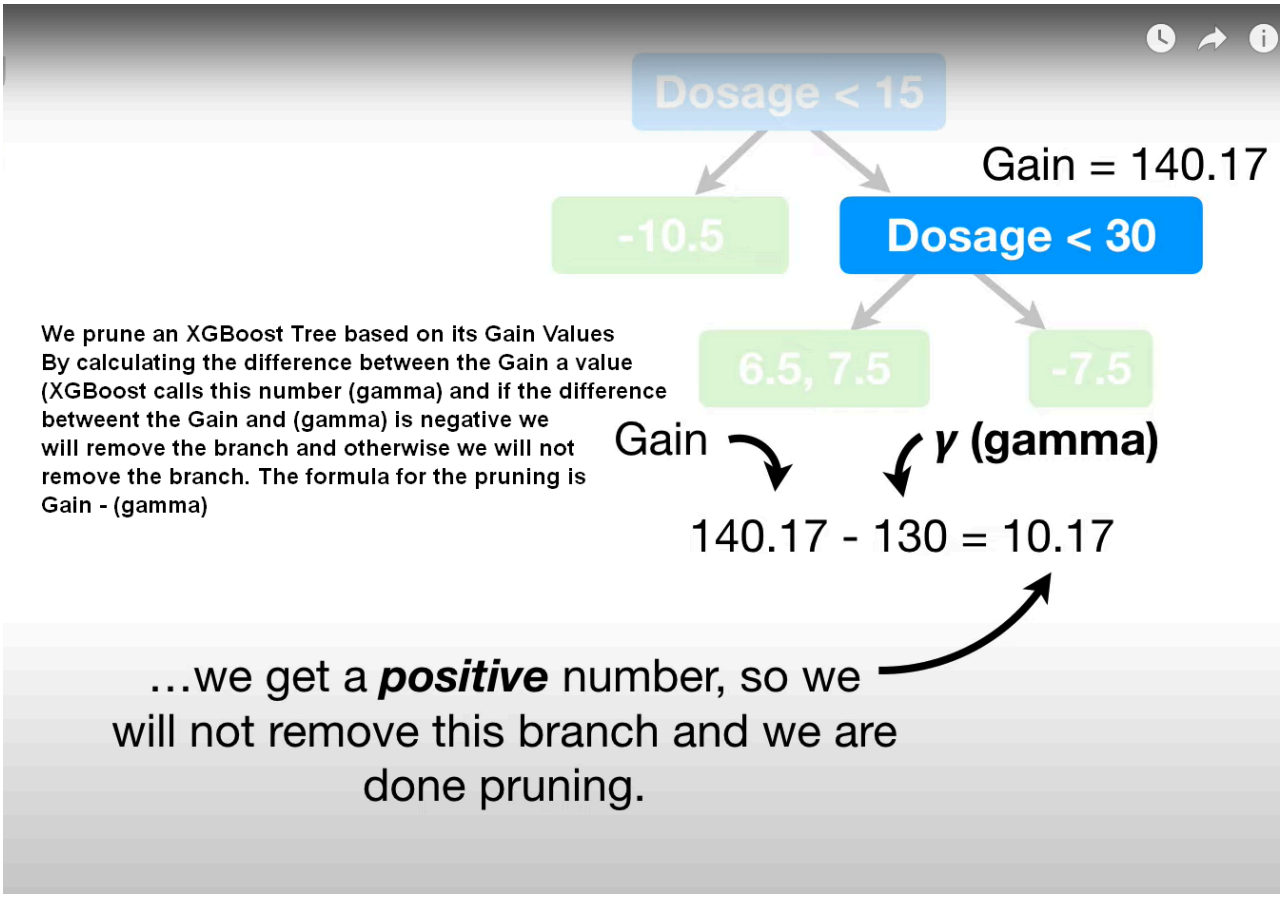
$Root_{Similarity}$ is parent leaf

For simplicity, each leaf can calculate its Similarity Score Splitting gain can be expressed as

Left(Similarity Score)+ Right(Similarity Score) - Parent(Similarity Score)

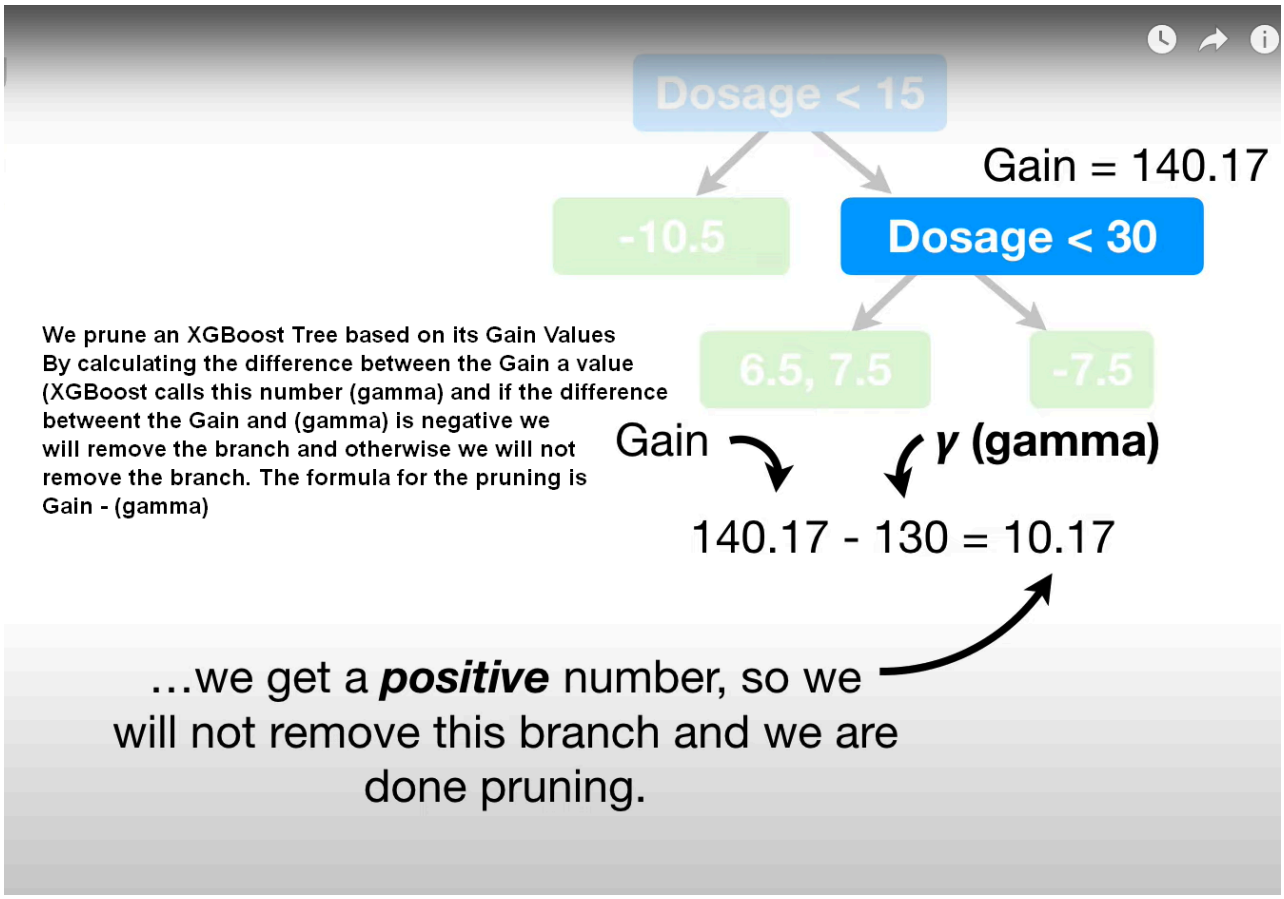
$$\text{Similarity Score} = \frac{\text{Sum of Residuals, Squared}}{\text{Number of Residuals} + \lambda}$$

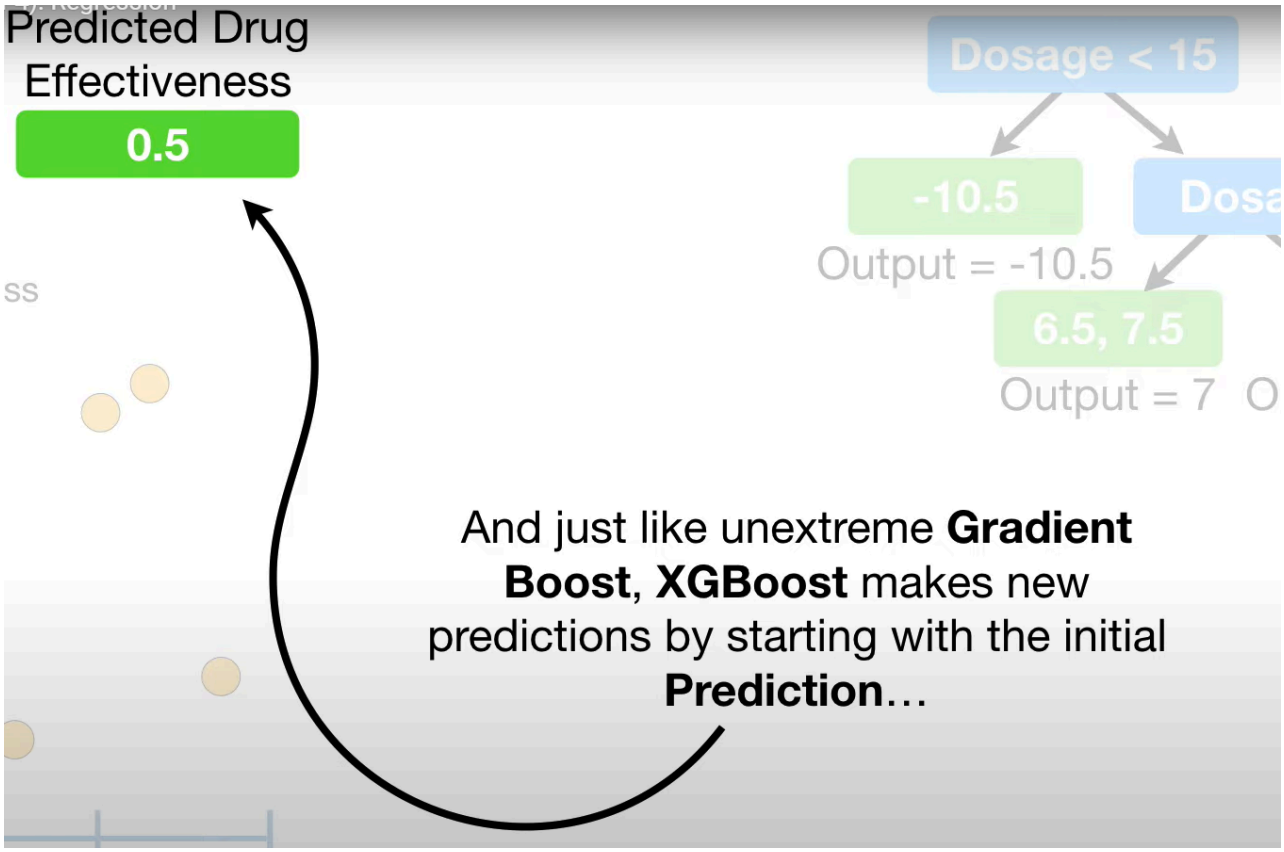
NOTE: λ (lambda) is a **Regularization** parameter, and we'll talk more about that later.

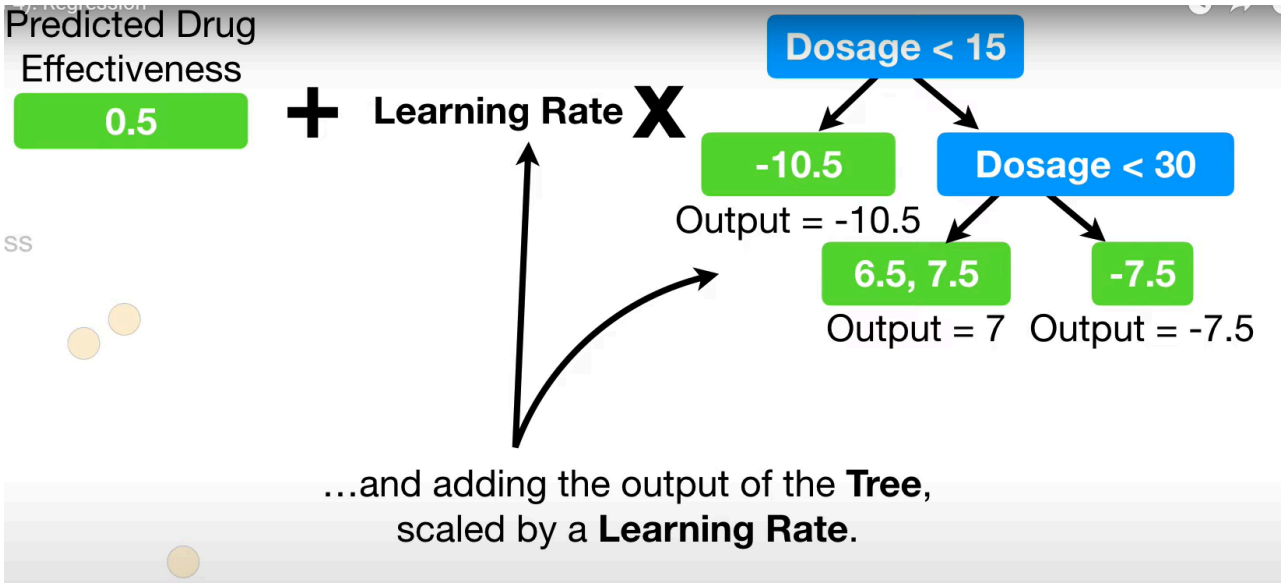


…we get a ***positive*** number, so we will not remove this branch and we are done pruning.

Thus, **λ (lambda)**, the **Regularization Parameter**, will reduce the prediction's sensitivity to this individual observation.







XGBoost calls the **Learning Rate**, **ε (eta)**, and the default value is **0.3**, so that's what we'll use.

…and that means a lower value for ***γ* (gamma)** will result in a negative difference and cause us to prune branches.

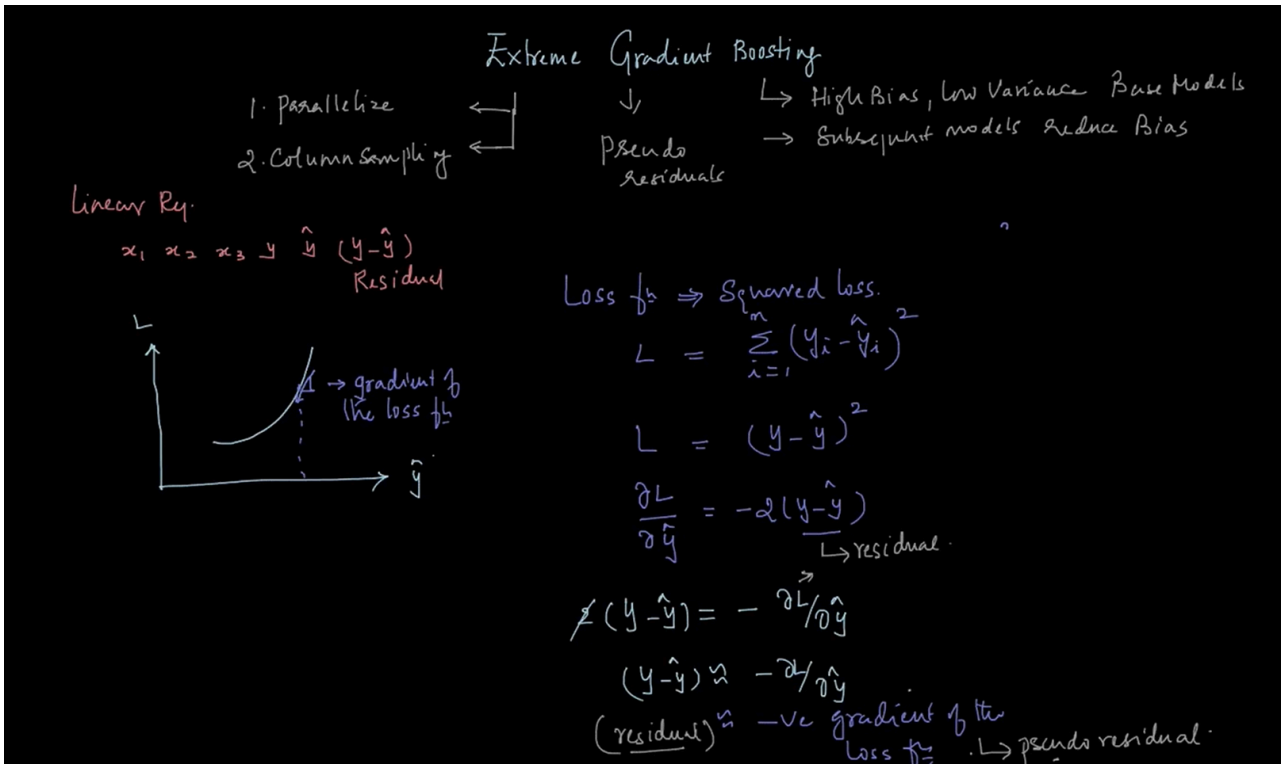
$$\text{Gain} - \gamma = \begin{cases} \text{If positive, then do not prune.} \\ \text{If negative, then prune.} \end{cases}$$

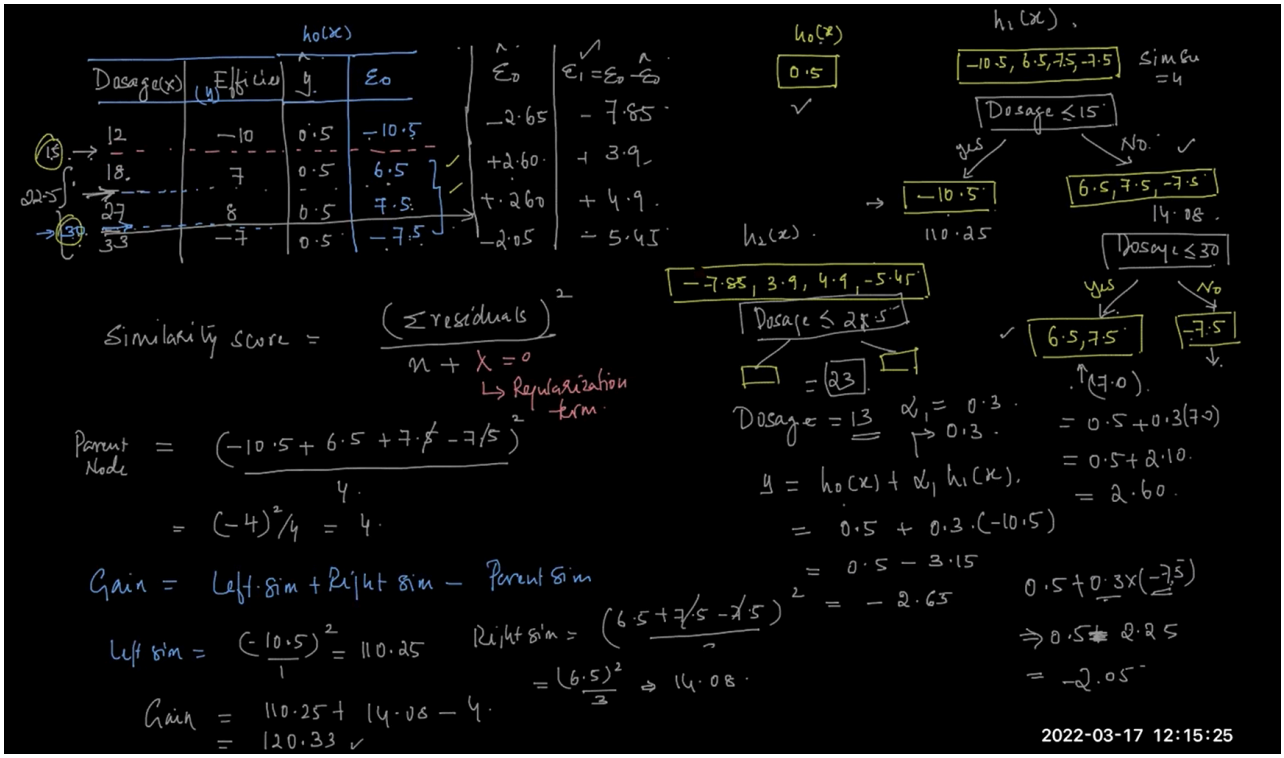
This formula converts probabilties into odds

$$\frac{p}{1 - p} = \text{odds}$$

$$\log\left(\frac{p}{1 - p}\right) = \log(\text{odds})$$

…we can get a formula that converts probabilities to the **log(odds)** by taking the log of both sides.

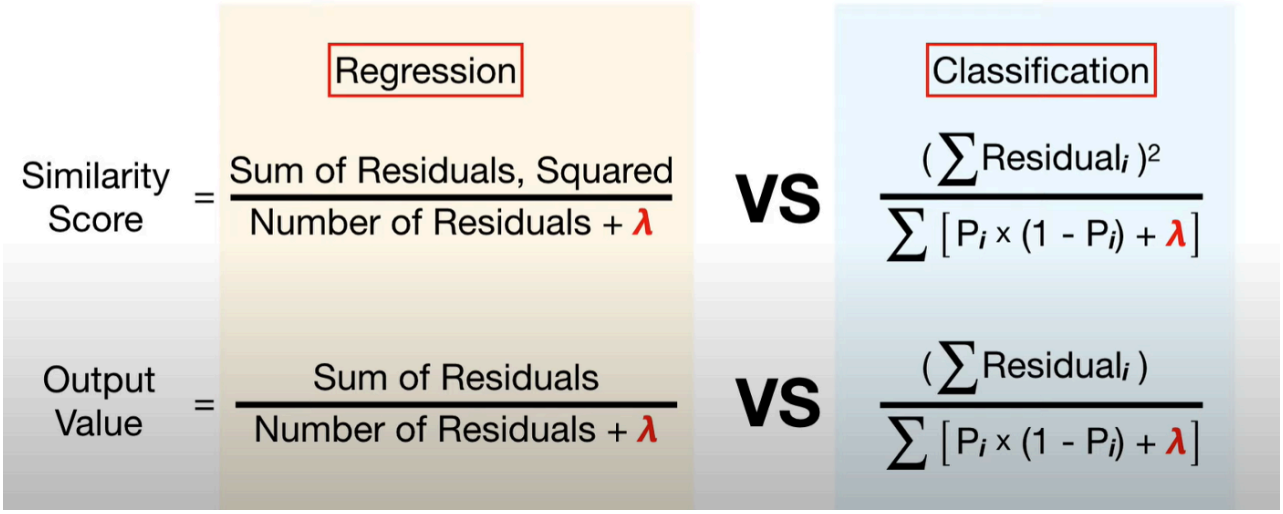




Simplified Summary For Regression and Classification

We know calculating tree node similarity and tree leaf output w_i will base on the chosen loss function, because g_i and h_i are 1-order and 2-order derivatives from loss function.

StatQuest with Josh Starmer gives a a good simplified summary for quick reference.



- Similarity Score is applied for every node in the tree
- Output Value normally is for leaf node output(w_i)

2.6 XGBoost Training Features

- When searching for best feature value for node split, XGBoost provides an option to search on the feature value's quantiles or histogram instead of try all the feature values to split node.
- When building feature histogram, XGBoost may split feature data into multiple computers to calculate histogram, then merge back to generate a aggregate histogram, this like Hadoop Map-reduce operation, and the generated histogram will be cached for next split.
- XGBoost can automatically handle missing values in feature. In tree node split step, XGBoost will either assign all missing value instances to left or right child, depend on which side has larger gain.
- XGBoost provide lots hyper-parameters to deal with overfitting

2.7 XGBoost Algorithm — Parameters

a. General Parameters Following are the General parameters used in Xgboost Algorithm:

- **booster:** The default value is GBtree. You need to specify the booster to use: GBtree (tree-based) or GBlinear (linear function).
- **num_pbuffer:** This is set automatically by XGBoost Algorithm, no need to be set by a user. Read the documentation of XGBoost for more details.
- num_feature:** This is set automatically by XGBoost Algorithm, no need to be set by a user.

b. Booster Parameters Below we discussed tree-specific parameters in Xgboost Algorithm:

- **eta:** The default value is set to 0.3. You need to specify step size shrinkage used in an update to prevents overfitting. After each boosting step, we can directly get the weights of new features. eta actually shrinks the feature weights to make the boosting process more conservative. The range is 0 to 1. Low eta value means the model is more robust to overfitting.
- **gamma:** The default value is set to 0. You need to specify the minimum loss reduction required to make a further partition on a leaf node of the tree. The larger, the more conservative the algorithm will be. The range is 0 to ∞. The larger the gamma more conservative the algorithm is.
- **max_depth:** The default value is set to 6. You need to specify the maximum depth of a tree. The range is 1 to ∞.
- **min_child_weight:** The default value is set to 1. You need to specify the minimum sum of instance weight(hessian) needed in a child. If the tree partition step results in a leaf node. Then with the sum of instance weight less than min_child_weight. Then the building process will give up further partitioning. In

linear regression mode, corresponds to a minimum number of instances needed to be in each node. The larger, the more conservative the algorithm will be. The range is 0 to ∞.

- **max_delta_step:** The default value is set to 0. Maximum delta step we allow each tree's weight estimation to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help make the update step more conservative. Usually, this parameter is not needed, but it might help in logistic regression. Especially, when a class is extremely imbalanced. Set it to a value of 1–10 might help control the update. The range is 0 to ∞.
- **subsample:** The default value is set to 1. You need to specify the subsample ratio of the training instance. Setting it to 0.5 means that XGBoost randomly collected half of the data instances. That needs to grow trees and this will prevent overfitting. The range is 0 to 1. colsample_bytree: The default value is set to 1. You need to specify the subsample ratio of columns when constructing each tree. The range is 0 to 1.

c. Linear Booster Specific Parameters These are Linear Booster Specific Parameters in XGBoost Algorithm.

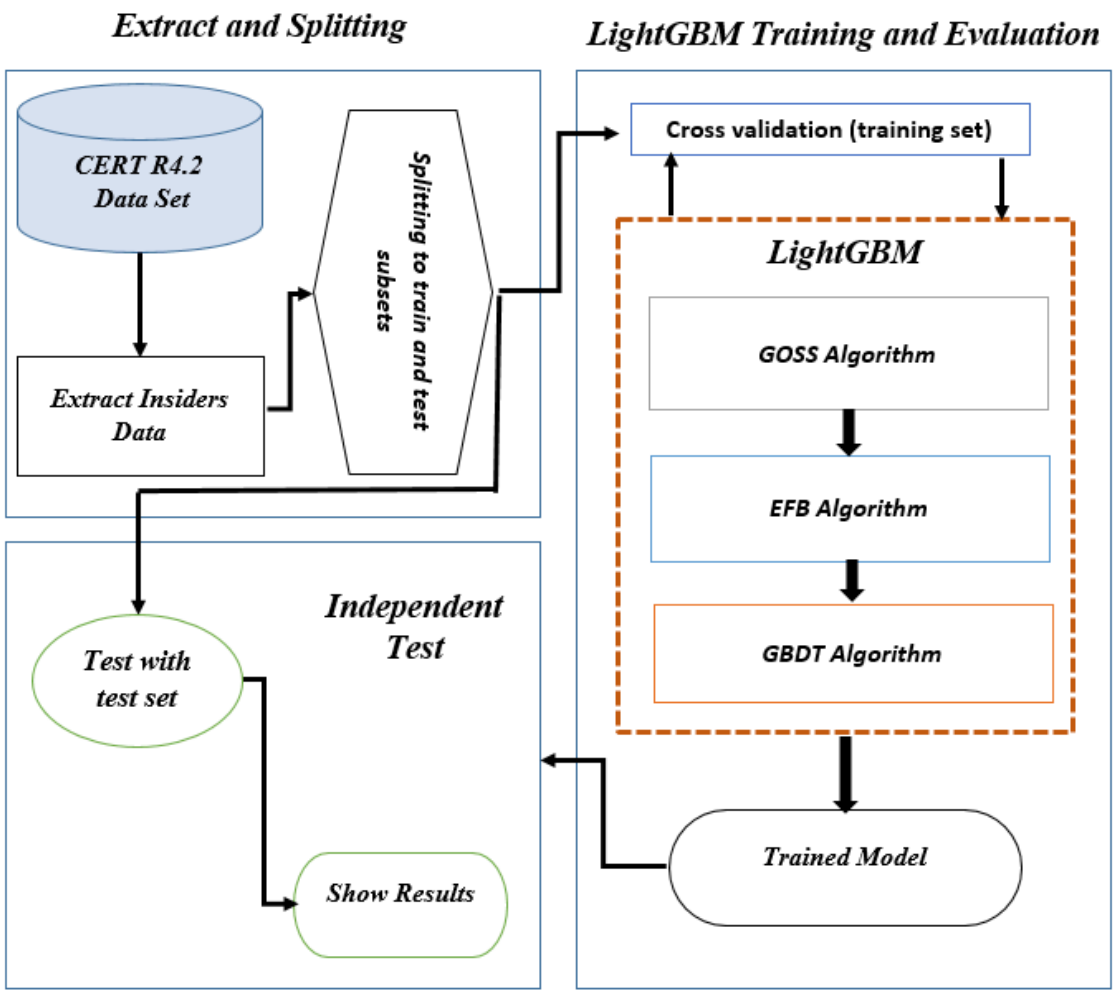
- **lambda and alpha:** These are regularization terms on weights. Lambda default value assumed is 1 and alpha is 0.
- **lambda_bias:** L2 regularization term on bias and has a default value of 0.

d. Learning Task Parameters Following are the Learning Task Parameters in XGBoost Algorithm

- **base_score:** The default value is set to 0.5. You need to specify the initial prediction score of all instances, global bias.
- **objective:** The default value is set to reg: linear. You need to specify the type of learner you want. That includes linear regression, Poisson regression, etc.
- **eval_metric:** You need to specify the evaluation metrics for validation data. And a default metric will be assigned according to the objective.
- **seed:** As always here you specify the seed to reproduce the same set of outputs.

Table of Content

3 Light Gradient Boosting Machine

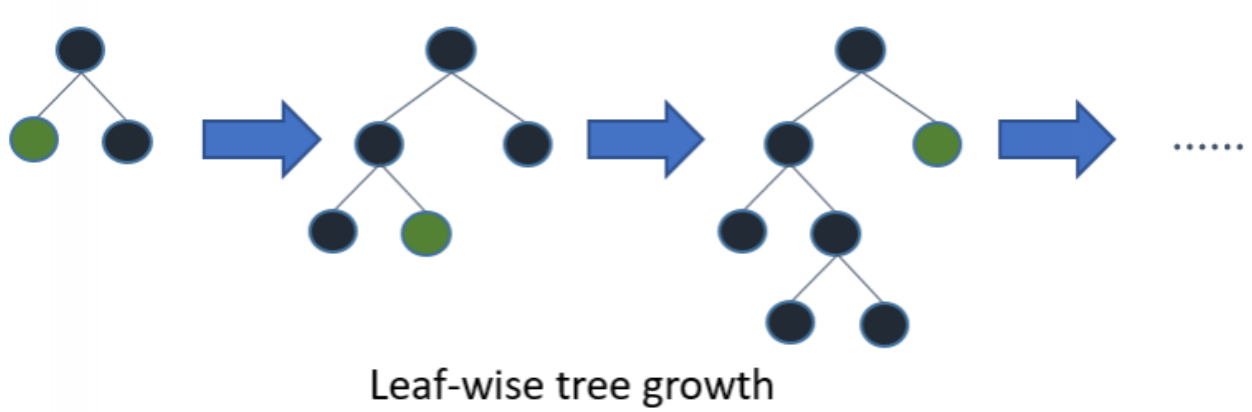


LightGBM extends the gradient boosting algorithm by adding a type of automatic feature selection as well as focusing on boosting examples with larger gradients. This can result in a dramatic speedup of training and improved predictive performance. LightGBM is able to handle huge amounts of data with ease. But keep in mind that this algorithm does not perform well with a small number of data points.

Let's take a moment to understand why that's the case.

The trees in LightGBM have a leaf-wise growth, rather than a level-wise growth. After the first split, the next split is done only on the leaf node that has a higher delta loss.

Consider the example I've illustrated in the below image:



Leaf-wise tree growth

After the first split, the left node had a higher loss and is selected for the next split. Now, we have three leaf nodes, and the middle leaf node had the highest loss. The leaf-wise split of the LightGBM algorithm enables it to work with large datasets.

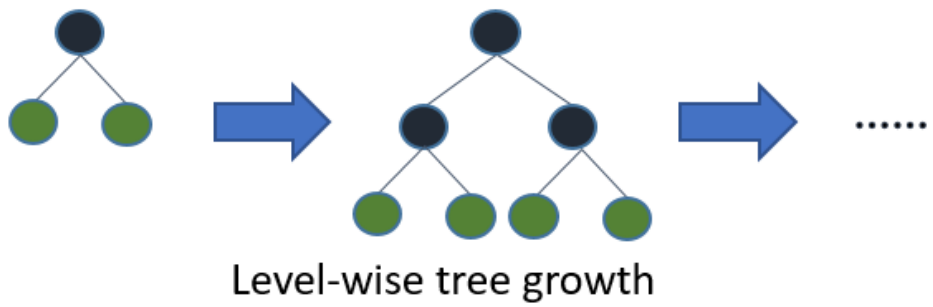
In order to speed up the training process, `LightGBM uses a histogram-based method for selecting the best split` . For any continuous variable, instead of using the individual values, these are divided into bins or buckets. This makes the training process faster and lowers memory usage.

Light GBM is a fast, distributed, high-performance gradient boosting framework based on decision tree algorithm, used for ranking, classification and many other machine learning tasks.

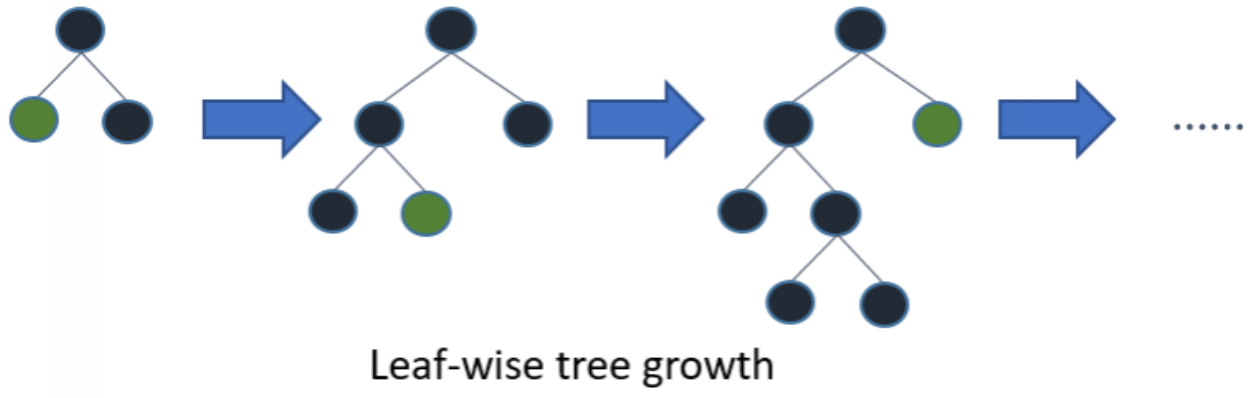
Since it is based on decision tree algorithms, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise. So when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms. Also, it is surprisingly very fast, hence the word 'Light'.

Before is a diagrammatic representation by the makers of the Light GBM to explain the difference clearly.

Level-wise tree growth in XGBOOST.



Leaf wise tree growth in Light GBM.



Leaf wise splits lead to increase in complexity and may lead to overfitting and it can be overcome by specifying another parameter max-depth which specifies the depth to which splitting will occur.

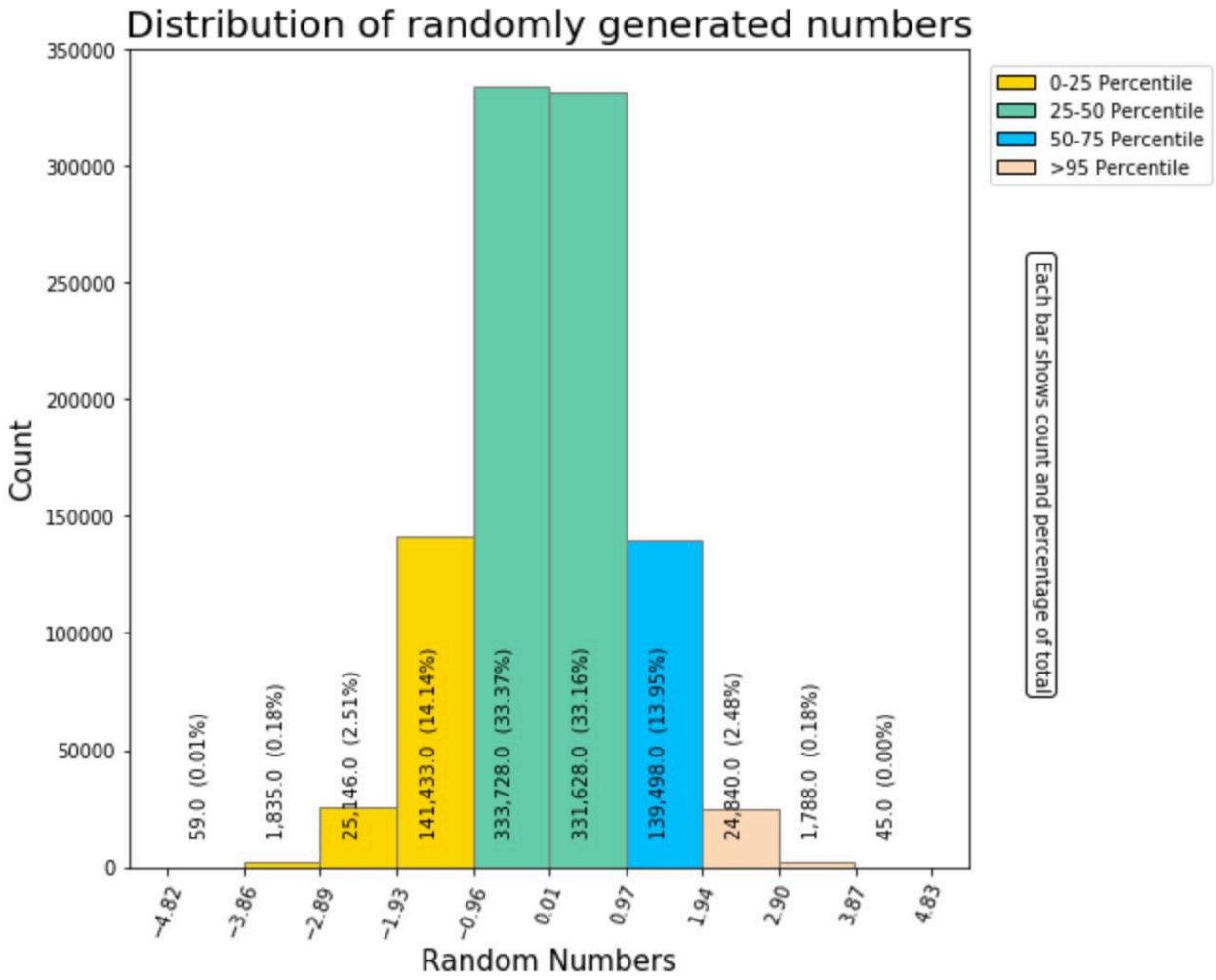
The costliest operation is training the decision tree and the most time consuming task is to find the optimum split points.

3.1 What are split points?

Split points are the feature values depending on which data is divided at a tree node. In the above example data division happens at node1 on Height (180) and at node 2 on Weight (80). The optimum splits are selected from a pool of candidate splits on the basis of information gain. In other words split points with maximum information gain are selected.

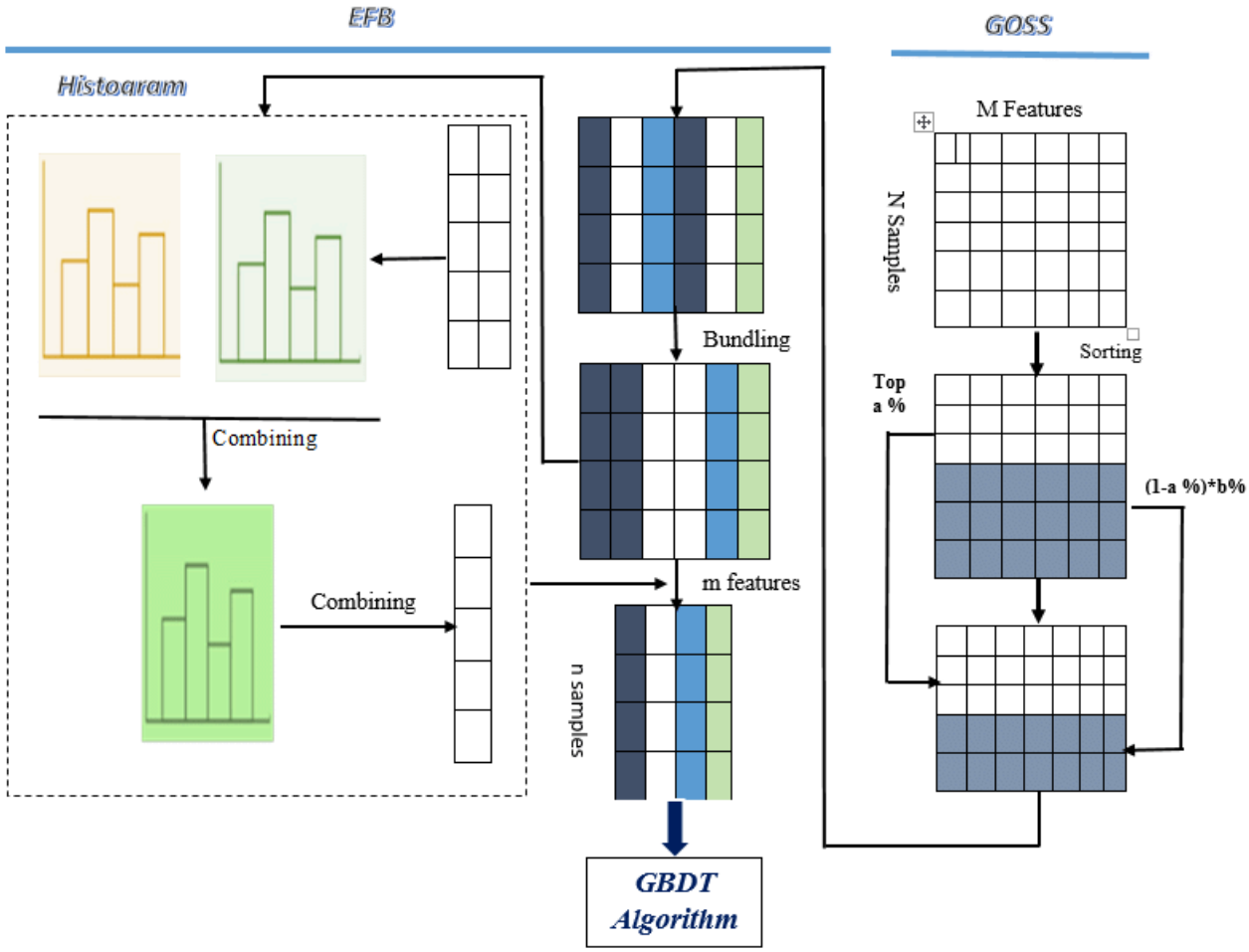
3.2 How are the optimum split points created?

Split finding algorithms are used to find candidate splits. One of the most popular split finding algorithm is the Pre-sorted algorithm which enumerates all possible split points on pre-sorted values. This method is simple but highly inefficient in terms of computation power and memory . The second method is the Histogram based algorithm which buckets continuous features into discrete bins to construct feature histograms during training. It costs O(#data * #feature) for histogram building and O(#bin * #feature) for split point finding. As bin << data histogram building will dominate the computational complexity.



What makes LightGBM special?

LightGBM aims to reduce complexity of histogram building (O(data * feature)) by down sampling data and feature using GOSS and EFB. What makes LightGBM different is that it uses a unique technique called Gradient-based One-Side Sampling (GOSS) to filter out the data instances to find a split value. This is different than XGBoost which uses pre-sorted and histogram-based algorithms to find the best split.



3.3 Structural Differences

Structural Differences in LightGBM & XGBoost LightGBM uses a novel technique of Gradient-based One-Side Sampling (GOSS) to filter out the data instances for finding a split value while XGBoost uses pre-sorted algorithm & Histogram-based algorithm for computing the best split. Here instances mean observations/samples. First, let us understand how pre-sorting splitting works- For each node, enumerate over all features For each feature, sort the instances by feature value Use a linear scan to decide the best split along that feature basis information gain Take the best split solution along all the features In simple terms, Histogram-based algorithm splits all the data points for a feature into discrete bins and uses these bins to find the split value of histogram. While, it is efficient than pre-sorted algorithm in training speed which enumerates all possible split points on the pre-sorted feature values, it is still behind GOSS in terms of speed. So what makes this GOSS method efficient? In AdaBoost, the sample weight serves as a good indicator for the importance of samples. However, in Gradient Boosting Decision Tree (GBDT), there are no native sample weights, and thus the sampling methods proposed for AdaBoost cannot be directly applied. Here comes gradient-basedsampling. Gradient represents the slope of the tangent of the loss function, so logically if gradient of data points are large in some sense, these points are important for finding the optimal split point as they have higher error GOSS keeps all the instances with large gradients and performs random sampling on the instances with small gradients.

For example, let's say I have 500K rows of data where 10k rows have higher gradients. So my algorithm will choose (10k rows of higher gradient+ x% of remaining 490k rows chosen randomly). Assuming x is 10%, total rows selected are 59k out of 500K on the basis of which split value if found. The basic assumption taken here is that samples with training instances with small gradients have smaller training error and it is already well-trained. In order to keep the same data distribution, when computing the information gain, GOSS introduces a constant multiplier for the data instances with small gradients. Thus, GOSS achieves a good balance between reducing the number of data instances and keeping the accuracy for learned decision trees.

3.4 What is GOSS?

Gradient-based One-Side Sampling, or GOSS for short, is a modification to the gradient boosting method that focuses attention on those training examples that result in a larger gradient, in turn speeding up learning and reducing the computational complexity of the method.

GOSS is a novel sampling method which down samples the instances on basis of gradients. As we know instances with small gradients are well trained (small training error) and those with large gradients are under trained. A naive approach to downsample is to discard instances with small gradients by solely focussing on instances with large gradients but this would alter the data distribution. In a nutshell GOSS retains instances with large gradients while performing random sampling on instances with small gradients.

With GOSS, we exclude a significant proportion of data instances with small gradients, and only use the rest to estimate the information gain. We prove that, since the data instances with larger gradients play a more important role in the computation of information gain, GOSS can obtain quite accurate estimation of the information gain with a much smaller data size.

Intuitive steps for GOSS calculation

1. Sort the instances according to absolute gradients in a descending order
2. Select the top a * 100% instances. [Under trained / large gradients]
3. Randomly samples b * 100% instances from the rest of the data. This will reduce the contribution of well trained examples by a factor of b (b < 1)
4. Without point 3 count of samples having small gradients would be 1-a (currently it is b). In order to maintain the original distribution LightGBM amplifies the contribution of samples having small gradients by a constant (1-a)/b to put more focus on the under-trained instances. This puts more focus on the under trained instances without changing the data distribution by much.

3.5 What is EFB(Exclusive Feature Bundling)?

Exclusive Feature Bundling, or EFB for short, is an approach for bundling sparse (mostly zero) mutually exclusive features, such as categorical variable inputs that have been one-hot encoded. As such, it is a type of automatic feature selection.

Remember histogram building takes O(#data * #feature). If we are able to down sample the #feature we will speed up tree learning. LightGBM achieves this by bundling features together. We generally work with high dimensionality data. Such data have many features which are mutually exclusive i.e they never take zero values simultaneously. LightGBM safely identifies such features and bundles them into a single feature to reduce the complexity to O(#data * #bundle) where #bundle << #feature.

Part 1 of EFB : Identifying features that could be bundled together

Intuitive explanation for creating feature bundles

Construct a graph with weighted (measure of conflict between features) edges. Conflict is measure of the fraction of exclusive features which have overlapping non zero values. Sort the features by count of non zero instances in descending order. Loop over the ordered list of features and assign the feature to an existing bundle (if conflict < threshold) or create a new bundle (if conflict > threshold).

Algorithm for merging features

We will try to understand the intuition behind merging features by an example. But before that let’s answer the following questions :

What is EFB achieving?

EFB is merging the features to reduce the training complexity. In order to keep the merge reversible we will keep exclusive features reside in different bins.

Example of the merge

In the example below you can see that feature1 and feature2 are mutually exclusive. In order to achieve non overlapping buckets we add bundle size of feature1 to feature2. This makes sure that non zero data points of bundled features (feature1 and feature2) reside in different buckets. In feature_bundle buckets 1 to 4 contains non zero instances of feature1 and buckets 5,6 contain non zero instances of feature2.

feature1	feature2	feature_bundle
0	2	6
0	1	5

Releases

No releases published

Packages

No packages published

Languages

- **Jupyter Notebook** 100.0%