Boosting

Bagging

Bagging build many independent predictors/models/learners and combine them using some model averaging techniques. (E.g. weighted average, majority vote or normal average).

We typically take random sub-sample/bootstrap of data for each model, so that all the models are little different from each other. Each observation is chosen with replacement to be used as input for each of the model. So, each model will have different observations based on the bootstrap process. Because this technique takes many uncorrelated learners to make a final model, it reduces error by reducing variance. Example of bagging ensemble is **Random Forest models.**

It is an approach where you take random samples of data, build learning algorithms and take simple means to find bagging probabilities.

Boosting

Boosting is an ensemble technique in which the predictors are not made independently, but sequentially.

Boosting is a method of converting weak learners into strong learners. The selection of sample is made more intelligently. We subsequently give more and more weight to hard to classify observations.

This technique employs the logic in which the subsequent predictors learn from the mistakes of the previous predictors. Therefore, the observations have an unequal probability of appearing in subsequent models and ones with the highest error appear most. (So, the observations are not chosen based on the bootstrap process, but based on the error). The predictors can be chosen from a range of models like decision trees, regressors, classifiers etc. Because new predictors are learning from mistakes committed by previous predictors, it takes less time/iterations to reach close to actual predictions. But we have to choose the stopping criteria carefully or it could lead to overfitting on training data.

Types of Boosting Algorithms

1. AdaBoost (**Ada**ptive **Boost**ing)
2. Gradient Boosting
3. XGBoost

AdaBoost

AdaBoost (Ada**ptive Boost**ing) it fits a sequence of weak learners on different weighted training data. It starts by predicting original data set and gives equal weight to each observation. If prediction is incorrect using the first learner, then it gives higher weight to observation which have been predicted incorrectly.

Being an iterative process, it continues to add learner(s) until a limit is reached in the number of models or accuracy.

A weak hypothesis or weak learner is defined as one whose performance is at least slightly better than random chance. The weak learners in AdaBoost are decision trees with a single split, called decision stumps for their shortness.

AdaBoost works by weighting the observations, putting more weight on difficult to classify instances and less on those already handled well. This means that samples that are difficult to classify receive increasing larger weights until the algorithm identifies a model that correctly classifies these samples. If there is any prediction error caused by first base learning algorithm, then we pay higher attention to observations having prediction error. Boosting pays higher focus on examples which are mis-classified or have higher errors by preceding weak rules. New weak learners are added sequentially that focus their training on the more difficult patterns.

Predictions are made by majority vote of the weak learners' predictions, weighted by their individual accuracy.

The AdaBoost Algorithm begins by training a decision tree in which each observation is assigned an equal weight. After evaluating the first tree, we increase the weights of those observations that are difficult to classify and lower the weights for those that are easy to classify. The second tree is therefore grown on this weighted data. Here, the idea is to improve upon the predictions of the first tree. Our new model is therefore Tree 1 + Tree 2. We then compute the classification error from this new 2-tree ensemble model and grow a third tree to predict the revised residuals. We repeat this process for a specified number of iterations. Subsequent trees help us to classify observations that are not well classified by the previous trees. Predictions of the final ensemble model is therefore the weighted sum of the predictions made by the previous tree models.

Adaboost helps you **combine multiple "weak classifiers" into a single "strong classifier"**. Here are some (fun) facts about Adaboost!
→ The weak learners in AdaBoost are decision trees with a single split, called decision stumps.
→ AdaBoost works by putting more weight on difficult to classify instances and less on those already handled well.
→ AdaBoost algorithms can be used for both classification and regression problem.
→ It predicts class between (-1, 1). Here -1 denotes the negative class while 1 represents the positive one. Y can only be -1 or 1.

Step 1: The base learner takes all the distributions and assign equal weight or attention to each observation.

Step *2*: If there is any prediction error caused by first base learning algorithm, then we pay higher attention to observations having prediction error. Then, we apply the next base learning algorithm.

Step *3*: Iterate Step 2 till the limit of base learning algorithm is reached or higher accuracy is achieved.

Finally, it combines the outputs from weak learner and creates a strong learner which eventually improves the prediction power of the model. Boosting pays higher focus on examples which are mis-classified or have higher errors by preceding weak rules.

Below are the steps for performing the AdaBoost algorithm:

1. Initially, all observations in the dataset are given equal weights.
2. A model is built on a subset of data.
3. Using this model, predictions are made on the whole dataset.
4. Errors are calculated by comparing the predictions and actual values.
5. While creating the next model, higher weights are given to the data points which were predicted incorrectly.
6. Weights can be determined using the error value. For instance, higher the error more is the weight assigned to the observation.
7. This process is repeated until the error function does not change, or the maximum limit of the number of estimators is reached.

Gradient Boosting

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). 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 (y=ax+b+e, e needs a special mention as it is the error term). The loss function is a measure indicating how good model's coefficients are at fitting the underlying data.

For example, if we are trying to predict the sales prices by using a regression, then the loss function would be based off the error between true and predicted house prices. Similarly, if our goal is to classify credit defaults, then the loss function would be a measure of how good our predictive model is at classifying bad loans.

Gradient Boosting or GBM is another ensemble machine learning algorithm that works for both regression and classification problems. GBM uses the boosting technique, combining a number of weak learners to form a strong learner. Regression trees used as a base learner, each subsequent tree in series is built on the errors calculated by the previous tree.

How Gradient Boosting work -1

We will use a simple example to understand the GBM algorithm. We have to predict the age of a group of people using the below data:

We will use a simple example to understand the GBM algorithm. We have to predict the age of a group of people using the below data:

ID	Married	Gender	Current City	Monthly Income	Age (target)
1	Y	M	A	51,000	35
2	N	F	B	25,000	24
3	Y	M	A	74,000	38
4	N	F	A	29,000	30
5	N	F	B	37,000	33

1. The mean age is assumed to be the predicted value for all observations in the dataset.
2. The errors are calculated using this mean prediction and actual values of age.

ID	Married	Gender	Current City	Monthly Income	Age (target)	Mean Age (prediction 1)	Residual 1
1	Y	M	A	51,000	35	32	3
2	N	F	B	25,000	24	32	-8
3	Y	M	A	74,000	38	32	6
4	N	F	A	29,000	30	32	-2
5	N	F	B	37,000	33	32	1

3. A tree model is created using the errors calculated above as target variable. Our objective is to find the best split to minimize the error.
4. The predictions by this model are combined with the predictions 1.

ID	Age (target)	Mean Age (prediction 1)	Residual 1 (new target)	Prediction 2	Combine (mean+pred2)
1	35	32	3	3	35
2	24	32	-8	-5	27
3	38	32	6	3	35
4	30	32	-2	-5	27
5	33	32	1	3	35

5. This value calculated above is the new prediction.
6. New errors are calculated using this predicted value and actual value.

ID	Age (target)	Mean Age (prediction 1)	Residual 1 (new target)	Prediction 2	Combine (mean+pred2)	Residual 2 (latest target)
1	35	32	3	3	35	0
2	24	32	-8	-5	27	-3
3	38	32	6	3	35	-3
4	30	32	-2	-5	27	3
5	33	32	1	3	35	-2

7. Steps 2 to 6 are repeated till the maximum number of iterations is reached (or error function does not change).

How Gradient Boosting work -2

In gradient 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. The final strong learner brings down both the bias and the variance.

Having a large number of trees might lead to overfitting. So, it is necessary to carefully choose the stopping criteria for boosting.

Boosting consists of three simple steps:

- An initial model F0 is defined to predict the target variable y. This model will be associated with a residual (y – F0)
- A new model h1 is fit to the residuals from the previous step
- Now, F0 and h1 are combined to give F1, the boosted version of F0. The mean squared error from F1 will be lower than that from F0:

$$F_1(x) <\text{-}\ F_0(x) + h_1(x)$$

To improve the performance of F1, we could model after the residuals of F1 and create a new model F2:

$$F_2(x) <\text{-}\ F_1(x) + h_2(x)$$

This can be done for *'m'* iterations, until residuals have been minimized as much as possible:

$$F_m(x) <\text{-}\ F_{m-1}(x) + h_m(x)$$

Here, the additive learners do not disturb the functions created in the previous steps. Instead, they impart information of their own to bring down the errors.

Demonstrating the Potential of Boosting

Consider the following data where the years of experience is predictor variable and salary (in thousand dollars) is the target. Using regression trees as base learners, we can create a model to predict the salary. For the sake of simplicity, we can choose square loss as our loss function and our objective would be to minimize the square error.

Years	Salary
5	82
7	80
12	103
23	118
25	172
28	127
29	204
34	189
35	99
40	166

As the first step, the model should be initialized with a function F0(x). F0(x) should be a function which minimizes the loss function or MSE (mean squared error), in this case:

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

$$argmin_\gamma \sum_{i=1}^{n} L(y_i, \gamma) = argmin_\gamma \sum_{i=1}^{n} (y_i - \gamma)^2$$

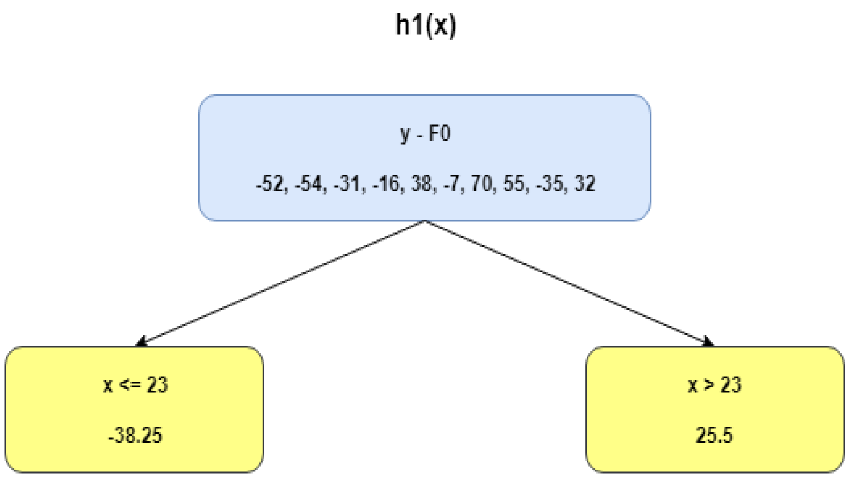
Taking the first differential of the above equation with respect to γ, it is seen that the function minimizes at the mean i=1nyin. So, the boosting model could be initiated with:

$$F_0(x) = \frac{\sum_{i=1}^{n} y_i}{n}$$

F0(x) gives the predictions from the first stage of our model. Now, the residual error for each instance is (yi – F0(x)).

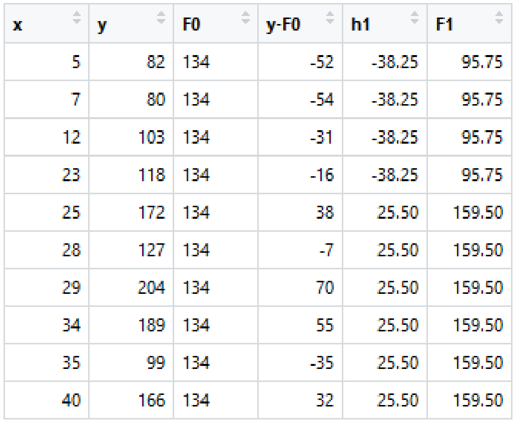
x	y	F0	y - F0
5	82	134	-52
7	80	134	-54
12	103	134	-31
23	118	134	-16
25	172	134	38
28	127	134	-7
29	204	134	70
34	189	134	55
35	99	134	-35
40	166	134	32

We can use the residuals from F0(x) to create h1(x). h1(x) will be a regression tree which will try and reduce the residuals from the previous step. The output of h1(x) won't be a prediction of y; instead, it will help in predicting the successive function F1(x) which will bring down the residuals.



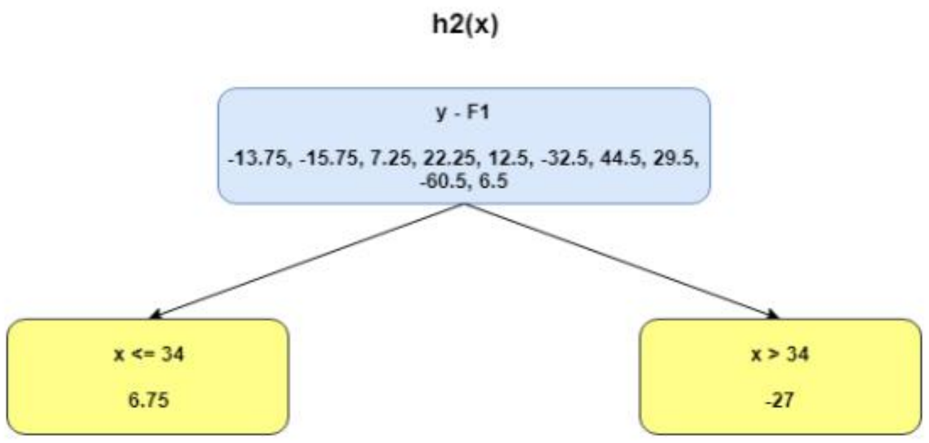
The additive model h1(x) computes the mean of the residuals (y – F0) at each leaf of the tree. The boosted function F1(x) is obtained by summing F0(x) and h1(x). This way h1(x) learns from the residuals of F0(x) and suppresses it in F1(x).

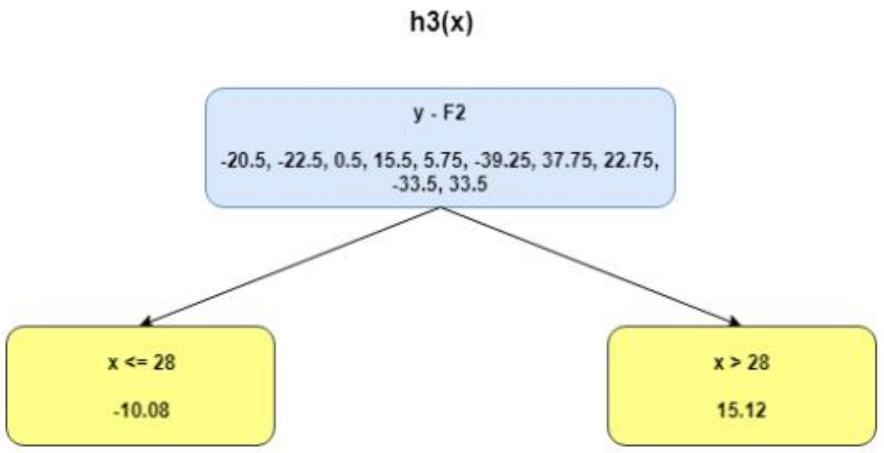
The additive model h1(x) computes the mean of the residuals (y – F0) at each leaf of the tree. The boosted function F1(x) is obtained by summing F0(x) and h1(x). This way h1(x) learns from the residuals of F0(x) and suppresses it in F1(x).

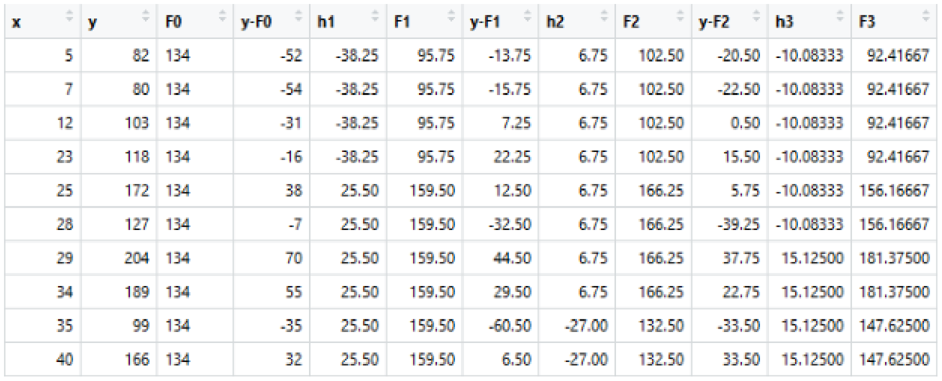


x	y	F0	y-F0	h1	F1
5	82	134	-52	-38.25	95.75
7	80	134	-54	-38.25	95.75
12	103	134	-31	-38.25	95.75
23	118	134	-16	-38.25	95.75
25	172	134	38	25.50	159.50
28	127	134	-7	25.50	159.50
29	204	134	70	25.50	159.50
34	189	134	55	25.50	159.50
35	99	134	-35	25.50	159.50
40	166	134	32	25.50	159.50

This can be repeated for 2 more iterations to compute h2(x) and h3(x). Each of these additive learners, hm(x), will make use of the residuals from the preceding function, Fm-1(x).







x	y	F0	y-F0	h1	F1	y-F1	h2	F2	y-F2	h3	F3
5	82	134	-52	-38.25	95.75	-13.75	6.75	102.50	-20.50	-10.08333	92.41667
7	80	134	-54	-38.25	95.75	-15.75	6.75	102.50	-22.50	-10.08333	92.41667
12	103	134	-31	-38.25	95.75	7.25	6.75	102.50	0.50	-10.08333	92.41667
23	118	134	-16	-38.25	95.75	22.25	6.75	102.50	15.50	-10.08333	92.41667
25	172	134	38	25.50	159.50	12.50	6.75	166.25	5.75	-10.08333	156.16667
28	127	134	-7	25.50	159.50	-32.50	6.75	166.25	-39.25	-10.08333	156.16667
29	204	134	70	25.50	159.50	44.50	6.75	166.25	37.75	15.12500	181.37500
34	189	134	55	25.50	159.50	29.50	6.75	166.25	22.75	15.12500	181.37500
35	99	134	-35	25.50	159.50	-60.50	-27.00	132.50	-33.50	15.12500	147.62500
40	166	134	32	25.50	159.50	6.50	-27.00	132.50	33.50	15.12500	147.62500

The MSEs for F0(x), F1(x) and F2(x) are 875, 692 and 540. It's amazing how these simple weak learners can bring about a huge reduction in error!

Note that each learner, hm(x), is trained on the residuals. All the additive learners in boosting are modeled after the residual errors at each step. Intuitively, it could be observed that the boosting learners make use of the patterns in residual errors. At the stage where maximum accuracy is reached by boosting, the residuals appear to be randomly distributed without any pattern.

Using gradient descent for optimizing the loss function

In the case discussed above, MSE was the loss function. The mean minimized the error here. When MAE (mean absolute error) is the loss function, the median would be used as F0(x) to initialize the model. A unit change in y would cause a unit change in MAE as well.

For MSE, the change observed would be roughly exponential. Instead of fitting hm(x) on the residuals, fitting it on the gradient of loss function, or the step along which loss occurs, would make this process generic and applicable across all loss functions.

Gradient descent helps us minimize any differentiable function. Earlier, the regression tree for hm(x) predicted the mean residual at each terminal node of the tree. In gradient boosting, the average gradient component would be computed.

For each node, there is a factor γ with which hm(x) is multiplied. This accounts for the difference in impact of each branch of the split. Gradient boosting helps in predicting the optimal gradient for the additive model, unlike classical gradient descent techniques which reduce error in the output at each iteration.

The following steps are involved in gradient boosting:

- F0(x) – with which we initialize the boosting algorithm – is to be defined:

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

- The gradient of the loss function is computed iteratively:

$$r_{im} = -\alpha\left[\frac{\partial(L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, \text{ where } \alpha \text{ is the learning rate}$$

- Each hm(x) is fit on the gradient obtained at each step
- The multiplicative factor γm for each terminal node is derived and the boosted model Fm(x) is defined:

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

Two most popular ensemble methods are **bagging** and **boosting**.

- **Bagging:** Training a bunch of individual models in a parallel way. Each model is trained by a random subset of the data
- **Boosting:** Training a bunch of individual models in a sequential way. Each individual model learns from mistakes made by the previous model.

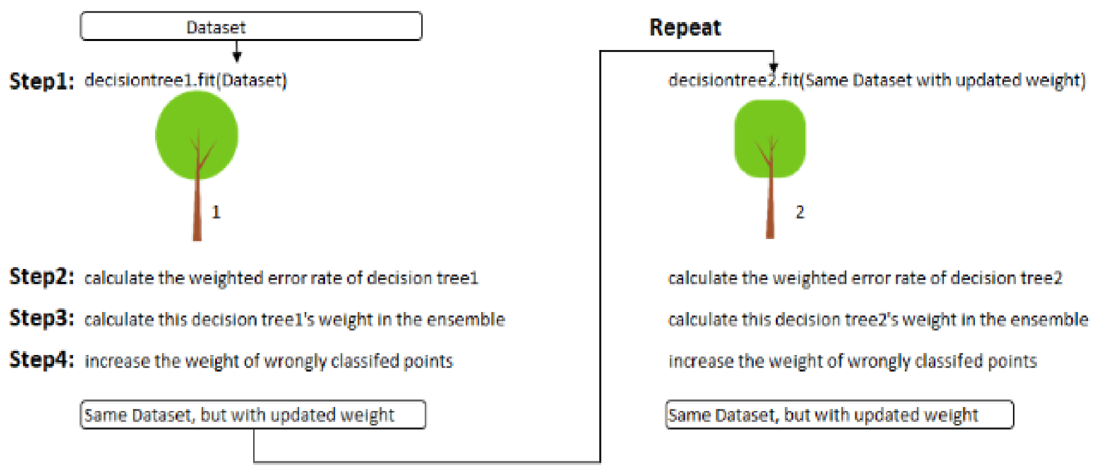
How Ada Boosting work -1

AdaBoost (Adaptive Boosting)

AdaBoost is a boosting ensemble model and works especially well with the decision tree. Boosting model's key is learning from the previous mistakes, e.g. misclassification data points.

AdaBoost learns from the mistakes by increasing the weight of misclassified data points.

Let's illustrate **how AdaBoost adapts**.



Dataset
Repeat
Step1: decisiontree1.fit(Dataset)
decisiontree2.fit(Same Dataset with updated weight)
1
2
Step2: calculate the weighted error rate of decision tree1
Step3: calculate this decision tree1's weight in the ensemble
Step4: increase the weight of wrongly classifed points
calculate the weighted error rate of decision tree2
calculate this decision tree2's weight in the ensemble
increase the weight of wrongly classifed points
Same Dataset, but with updated weight
Same Dataset, but with updated weight

Step 0: **Initialize the weights** of data points. if the training set has 100 data points, then each point's initial weight should be 1/100 = 0.01.

Step 1: **Train** a decision tree

Step 2: **Calculate the weighted error rate (e)** of the decision tree. **The weighted error rate (e)** is just how many wrong predictions out of total and you treat the wrong predictions differently based on its data point's weight. **The higher the weight, the more the corresponding error will be weighted** during the calculation of the (e).

Step 3: **Calculate this decision tree's weight** in the ensemble

the weight of this tree = learning rate * log((1 — e) / e)

- the higher weighted error rate of a tree, 😫, the less decision power the tree will be given during the later voting
- the lower weighted error rate of a tree, 😃, the higher decision power the tree will be given during the later voting

Step 4: **Update weights** of wrongly classified points

the weight of each data point =

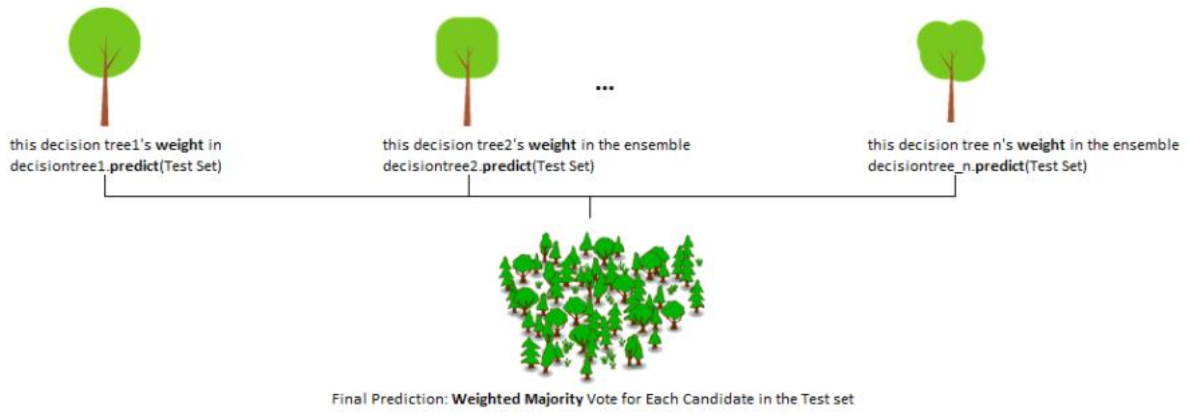
- if the model got this data point correct, the weight stays the same
- if the model got this data point wrong, the new weight of this point = old weight * np.exp(weight of this tree)

Note: The higher the weight of the tree (more accurate this tree performs), the more boost (importance) the misclassified data point by this tree will get. The weights of the data points are normalized after all the misclassified points are updated.

Step 5: **Repeat** Step 1(until the number of trees we set to train is reached)

Step 6: **Make the final prediction**

The AdaBoost makes a new prediction by adding up the weight (of each tree) multiply the prediction (of each tree). Obviously, the tree with higher weight will have more power of influence the final decision.


this decision tree1's **weight** in
decisiontree1.**predict**(Test Set)
this decision tree2's **weight** in the ensemble
decisiontree2.**predict**(Test Set)
...
this decision tree n's **weight** in the ensemble
decisiontree_n.**predict**(Test Set)
Final Prediction: **Weighted Majority** Vote for Each Candidate in the Test set

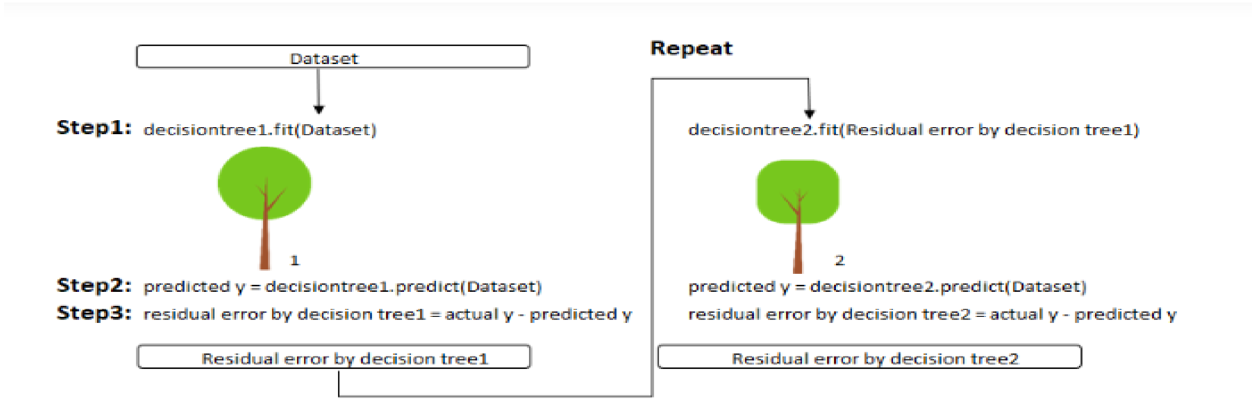
How Gradient Boosting work -3

Gradient Boosting

Gradient boosting is another boosting model. Remember, boosting model's key is learning from the previous mistakes.

Gradient Boosting learns from the mistake — residual error directly, rather than update the weights of data points.

Let's illustrate **how Gradient Boost learns.**


Dataset
Repeat
Step1: decisiontree1.fit(Dataset)
decisiontree2.fit(Residual error by decision tree1)
1
2
Step2: predicted y = decisiontree1.predict(Dataset)
Step3: residual error by decision tree1 = actual y - predicted y
predicted y = decisiontree2.predict(Dataset)
residual error by decision tree2 = actual y - predicted y
Residual error by decision tree1
Residual error by decision tree2

Step 1: **Train** a decision tree

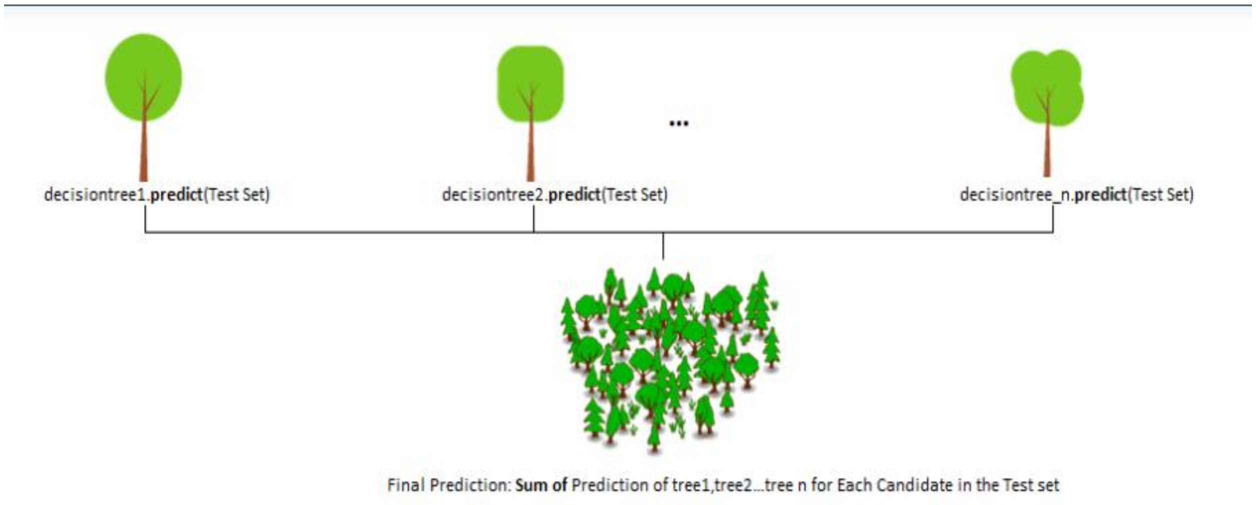
Step 2: **Apply** the decision tree just trained to predict

Step 3: **Calculate** the residual of this decision tree, Save residual errors as the new y

Step 4: **Repeat** Step 1 (until the number of trees we set to train is reached)

Step 5: **Make the final prediction**

The Gradient Boosting makes a new prediction by simply adding up the predictions (of all trees).


decisiontree1.predict(Test Set)
decisiontree2.predict(Test Set)
...
decisiontree_n.predict(Test Set)
Final Prediction: Sum of Prediction of tree1,tree2...tree n for Each Candidate in the Test set

How Gradient Boosting work -4

The example aims to predict salary per month (in dollars) based on whether or not the observation has own house, own car and own family/children. Suppose we have a dataset of three observations where the first variable is 'have own house', the second is 'have own car' and the third variable is 'have family/children', and target is 'salary per month'. The observations are

1.- (Yes,Yes, Yes, 10000)

2.-(No, No, No, 25)

3.-(Yes,No,No,5000)

Choose a number M of boosting stages, say $M = 1$. The first step of gradient boosting algorithm is to start with an initial model F_0. This model is a constant defined by $\mathrm{argmin}_\gamma \sum_{i=1}^{3} L(y_i, \gamma)$ in our case, where L is the loss function. Suppose that we are working with the usual loss function $L(y_i, \gamma) = \frac{1}{2}(y_i - \gamma)^2$. When this is the case, this constant is equal to the mean of the outputs y_i, so in our case $\frac{10000+25+5000}{3} = 5008.3$. So our initial model is $F_0(x) = 5008.3$ (which maps every observation x (e.g. (No,Yes,No)) to 5008.3.

Next we should create a new dataset, which is the previous dataset but instead of y_i we take the residuals $r_{i0} = -\frac{\partial L(y_i, F_0(x_i))}{\partial F_0(x_i)}$. In our case, we have $r_{i0} = y_i - F_0(x_i) = y_i - 5008.3$. So our dataset becomes

1.- (Yes,Yes, Yes, 4991.6)

2.-(No, No, No, -4983.3)

3.-(Yes,No,No,-8.3)

The next step is to fit a base learner h to this new dataset. Usually the base learner is a decision tree, so we use this.

Now assume that we constructed the following decision tree h. I constructed this tree using entropy and information gain formulas but probably I made some mistake, however for our purposes we can assume it's correct. For a more detailed example, please check

https://www.saedsayad.com/decision_tree.htm

The constructed tree is:

The constructed tree is:

$$\begin{tikzcd}
\text{have own house} \arrow[d] \arrow[dr] & \\
\text{Yes} \arrow[d] & \text{No} \arrow[d] \\
\text{have own car} \arrow[d] \arrow[dr] & \text{salary residual} = -4983.3 \\
\text{yes} \arrow[d] & \text{No} \arrow[d] \\
\text{salary residual} = 4991.6 & \text{salary residual} = -8.3
\end{tikzcd}$$

Let's call this decision tree h_0. The next step is to find a constant $\lambda_0 = \arg\min_\lambda \sum_{i=1}^{3} L(y_i, F_0(x_i) + \lambda h_0(x_i))$. Therefore, we want a constant λ minimizing

$$C = \tfrac{1}{2}(10000 - (5008.3 + \lambda * 4991.6))^2 + \tfrac{1}{2}(25 - (5008.3 + \lambda(-4983.3)))^2 + \tfrac{1}{2}(5000 - (5008.3 + \lambda(-8.3)))^2$$

This is where gradient descent comes in handy.

Suppose that we start at $P_0 = 0$. Choose the learning rate equal to $\eta = 0.01$. We have

$$\frac{\partial C}{\partial \lambda} = (10000 - (5008.3 + \lambda * 4991.6))(-4991.6) + (25 - (5008.3 + \lambda(-4983.3))) * 4983.3 + (5000 - (5008.3 + \lambda(-8.3))) * 8.3$$

Then our next value P_1 is given by
$P_1 = 0 - \eta \frac{\partial C}{\partial \lambda}(0) = 0 - .01(-4991.6 * 4991.7 + 4983.4 * (-4983.3) + (-8.3) * 8.3)$.

Repeat this step N times, and suppose that the last value is P_N. If N is sufficiently large and η is sufficiently small then $\lambda := P_N$ should be the value where $\sum_{i=1}^{3} L(y_i, F_0(x_i) + \lambda h_0(x_i))$ is minimized. If this is the case, then our λ_0 will be equal to P_N. Just for the sake of it, suppose that $P_N = 0.5$ (so that $\sum_{i=1}^{3} L(y_i, F_0(x_i) + \lambda h_0(x_i))$ is minimized at $\lambda := 0.5$). Therefore, $\lambda_0 = 0.5$.

The next step is to update our initial model F_0 by $F_1(x) := F_0(x) + \lambda_0 h_0(x)$. Since our number of boosting stages is just one, then this is our final model F_1.

Now suppose that I want to predict a new observation $x =$(Yes,Yes,No) (so this person does have own house and own car but no children). What is the salary per month of this person? We just compute $F_1(x) = F_0(x) + \lambda_0 h_0(x) = 5008.3 + 0.5 * 4991.6 = 7504.1$. So this person earns \$7504.1 per month according to our model.

Gradient Boosting algorithm

> *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:

$$Loss = MSE = \sum (y_i - y_i^p)^2$$

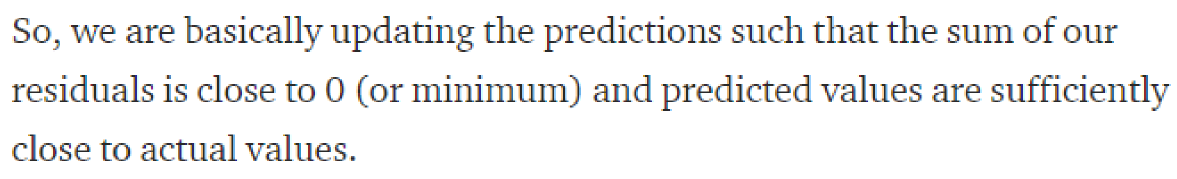
where, y_i = ith target value, y_i^p = ith prediction, $L(y_i, y_i^p)$ is Loss function

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.

$$y_i^p = y_i^p + \alpha * \delta \sum (y_i - y_i^p)^2 / \delta y_i^p$$

which becomes, $y_i^p = y_i^p - \alpha * 2 * \sum (y_i - y_i^p)$

where, α is learning rate and $\sum (y_i - y_i^p)$ is sum of residuals


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

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.

In summary,
- we first model data with simple models and analyze data for errors.
- These errors signify data points that are difficult to fit by a simple model.
- Then for later models, we particularly focus on those hard to fit data to get them right.
- In the end, we combine all the predictors by giving some weights to each predictor.

How Gradient Boosting work -5

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.

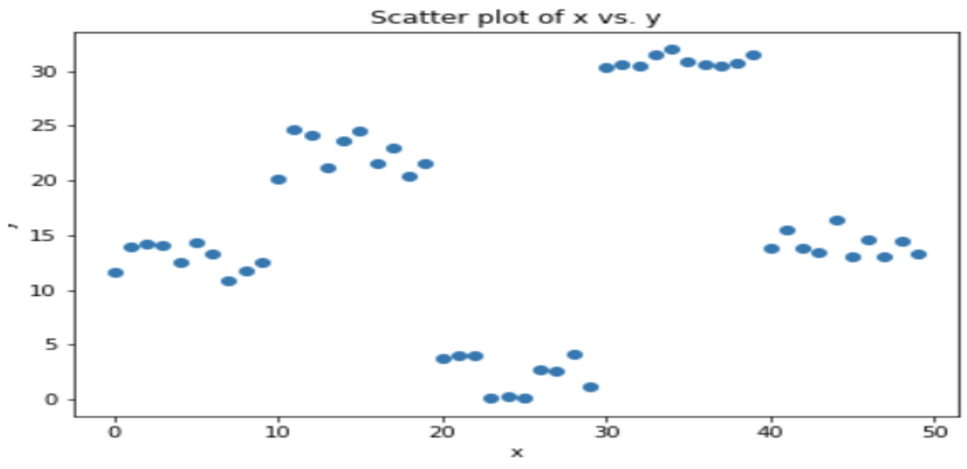


Fig 4. Simulated data (x: input, y: output)

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. **2.** Calculate error residuals. Actual target value, minus predicted target value **[e1= y - y_predicted1]**
3. **3.** Fit a new model on error residuals as target variable with same input variables **[call it e1_predicted]**
4. **4.** Add the predicted residuals to the previous predictions **[y_predicted2 = y_predicted1 + e1_predicted]**
5. **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 residuals become constant. Overfitting can be controlled by consistently checking accuracy on validation data.

Maths behind this logic

$$Predictions = y_i^p$$
$$Loss = L(y_i, y_i^p)$$
$$Loss = MSE = \sum (y_i - y_i^p)^2$$
$$y_i^p = y_i^p + \alpha * \delta L(y_i, y_i^p)/\delta y_i^p$$
$$y_i^p = y_i^p + \alpha * \delta \sum (y_i - y_i^p)^2/\delta y_i^p$$
$$y_i^p = y_i^p - \alpha * 2 * \sum (y_i - y_i^p)$$

where, y_i = ith target value, y_i^p = ith prediction, $L(y_i, y_i^p)$ is Loss function, α is learning rate. So the last equation tells us that, we need to adjust predictions based on our residuals, i.e. $\sum (y_i - y_i^p)$. This is what we did, we adjusted our predictions using the fit on residuals. (accordingly adjusting value of α

Ensemble Learning

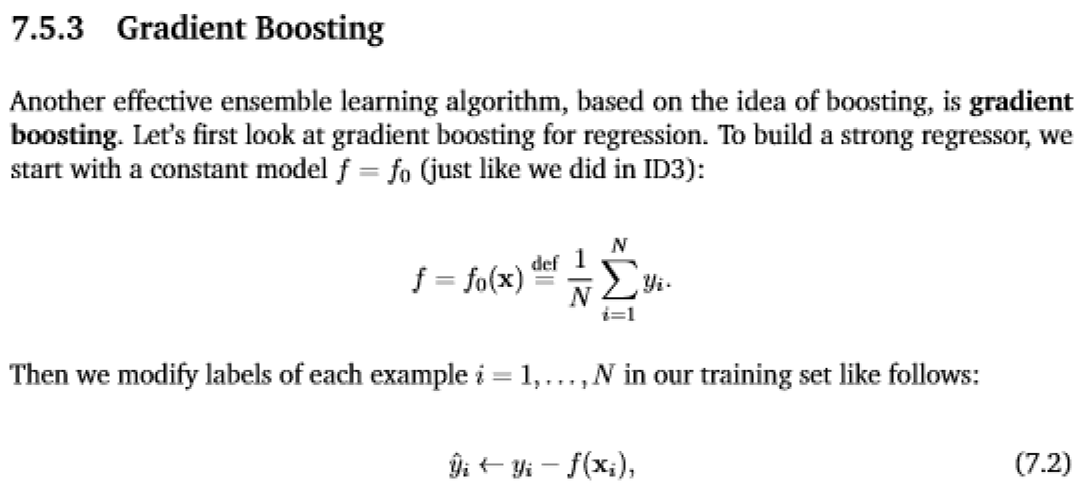
Ensemble learning is a learning paradigm that, instead of trying to learn one super-accurate model, focuses on training a large number of low-accuracy models and then combining the predictions given by those weak models to obtain a high-accuracy meta-model.

Low-accuracy models are usually learned by weak learners, that is learning algorithms that cannot learn complex models, and thus are typically fast at the training and at the prediction time. The most frequently used weak learner is a decision tree learning algorithm in which we often stop splitting the training set after just a few iterations. The obtained trees are shallow and not particularly accurate, but the idea behind ensemble learning is that if the trees are not identical and each tree is at least slightly better than random guessing, then we can obtain high accuracy by combining a large number of such trees. To obtain the prediction for input x, the predictions of each weak model are combined using some sort of weighted voting.

The specific form of vote weighting depends on the algorithm, but, independently of the algorithm, the idea is the same: if the council of weak models predicts that the message is spam, then we assign the label spam to x. Two principal ensemble learning methods are boosting and bagging.

Boosting consists of using the original training data and iteratively create multiple models by using a weak learner. Each new model would be different from the previous ones in the sense that the weak learner, by building each new model tries to “fix” the errors which previous models make. The final ensemble model is a certain combination of those multiple weak models built iteratively.

Bagging consists of creating many “copies” of the training data (each copy is slightly different from another) and then apply the weak learner to each copy to obtain multiple weak models and then combine them. A widely used and effective machine learning algorithm based on the idea of bagging is random forest.



7.5.3 Gradient Boosting

Another effective ensemble learning algorithm, based on the idea of boosting, is **gradient boosting**. Let’s first look at gradient boosting for regression. To build a strong regressor, we start with a constant model $f = f_0$ (just like we did in ID3):

$$f = f_0(\mathbf{x}) \overset{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N} y_i.$$

Then we modify labels of each example $i = 1, \ldots, N$ in our training set like follows:

$$\hat{y}_i \leftarrow y_i - f(\mathbf{x}_i), \tag{7.2}$$

where $\hat{y}_i$, called the **residual**, is the new label for example $\mathbf{x}_i$.

Now we use the modified training set, with residuals instead of original labels, to build a new decision tree model, f_1. The boosting model is now defined as $f \stackrel{\text{def}}{=} f_0 + \alpha f_1$, where α is the learning rate (a hyperparameter).

Then we recompute the residuals using eq. 7.2 and replace the labels in the training data once again, train the new decision tree model f_2, redefine the boosting model as $f \stackrel{\text{def}}{=} f_0 + \alpha f_1 + \alpha f_2$ and the process continues until the predefined maximum M of trees are combined.

Intuitively, what's happening here? By computing the residuals, we find how well (or poorly) the target of each training example is predicted by the current model f. We then train another tree to fix the errors of the current model (this is why we use residuals instead of real labels) and add this new tree to the existing model with some weight α. Therefore, each additional tree added to the model partially fixes the errors made by the previous trees until the maximum number M (another hyperparameter) of trees are combined.

Now you should reasonably ask why the algorithm is called *gradient* boosting? In gradient boosting, we don't calculate any gradient contrary to what we did in Chapter 4 for linear regression. To see the similarity between gradient boosting and gradient descent remember why we calculated the gradient in linear regression: we did that to get an idea on where we should move the values of our parameters so that the MSE cost function reaches its minimum. The gradient showed the direction, but we didn't know how far we should go in this direction, so we used a small step α at each iteration and then reevaluated our direction. The same happens in gradient boosting. However, instead of getting the gradient directly, we use its proxy in the form of residuals: they show us how the model has to be adjusted so that the error (the residual) is reduced.

The three principal hyperparameters to tune in gradient boosting are the number of trees, the learning rate, and the depth of trees — all three affect model accuracy. The depth of trees also affects the speed of training and prediction: the shorter, the faster.

It can be shown that training on residuals optimizes the overall model f for the mean squared error criterion. You can see the difference with bagging here: boosting reduces the bias (or underfitting) instead of the variance. As such, boosting can overfit. However, by tuning the depth and the number of trees, overfitting can be largely avoided.

Gradient boosting for classification is similar, but the steps are slightly different. Let's consider the binary case. Assume we have M regression decision trees. Similarly to logistic regression, the prediction of the ensemble of decision trees is modeled using the sigmoid function:

$$\Pr(y = 1|\mathbf{x}, f) \stackrel{\text{def}}{=} \frac{1}{1 + e^{-f(\mathbf{x})}},$$

where $f(\mathbf{x}) \stackrel{\text{def}}{=} \sum_{m=1}^{M} f_m(\mathbf{x})$ and f_m is a regression tree.

Again, like in logistic regression, we apply the maximum likelihood principle by trying to find such an f that maximizes $L_f = \sum_{i=1}^{N} \ln\left[\Pr(y_i = 1|\mathbf{x}_i, f)\right]$. Again, to avoid numerical overflow, we maximize the sum of log-likelihoods rather than the product of likelihoods.

The algorithm starts with the initial constant model $f = f_0 = \frac{p}{1-p}$, where $p = \frac{1}{N}\sum_{i=1}^{N} y_i$. (It can be shown that such initialization is optimal for the sigmoid function.) Then at each iteration m, a new tree f_m is added to the model. To find the best f_m, first the partial derivative g_i of the current model is calculated for each $i = 1, \ldots, N$:

$$g_i = \frac{dL_f}{df},$$

where f is the ensemble classifier model built at the previous iteration $m-1$. To calculate g_i we need to find the derivatives of $\ln\left[\Pr(y_i = 1|\mathbf{x}_i, f)\right]$ with respect to f for all i. Notice that $\ln\left[\Pr(y_i = 1|\mathbf{x}_i, f)\right] \stackrel{\text{def}}{=} \ln\left[\frac{1}{1+e^{-f(\mathbf{x}_i)}}\right]$. The derivative of the right-hand term in the previous equation with respect to f equals to $\frac{1}{e^{f(\mathbf{x}_i)}+1}$.

We then transform our training set by replacing the original label y_i with the corresponding partial derivative g_i, and build a new tree f_m using the transformed training set. Then we find the optimal update step ρ_m as:

$$\rho_m \leftarrow \arg\max_{\rho} L_{f+\rho f_m}.$$

At the end of iteration m, we update the ensemble model f by adding the new tree f_m:

$$f \leftarrow f + \alpha \rho_m f_m.$$

We iterate until $m = M$, then we stop and return the ensemble model f.

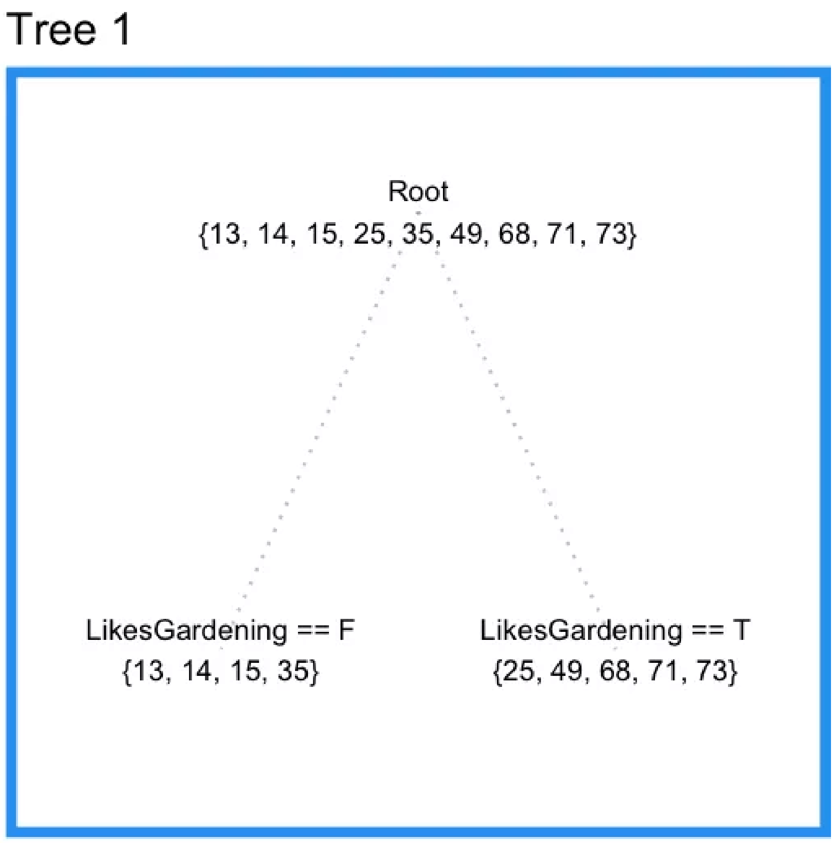
Gradient boosting is one of the most powerful machines learning algorithms. Not just because it creates very accurate models, but also because it is capable of handling huge datasets with millions of examples and features. It usually outperforms random forest in accuracy but, because of its sequential nature, can be significantly slower in training.

How Gradient Boosting work -6

We'll start with a simple example. We want to predict a person's age based on whether they play video games, enjoy gardening, and their preference on wearing hats. Our objective is to minimize squared error. We have these nine training samples to build our model.

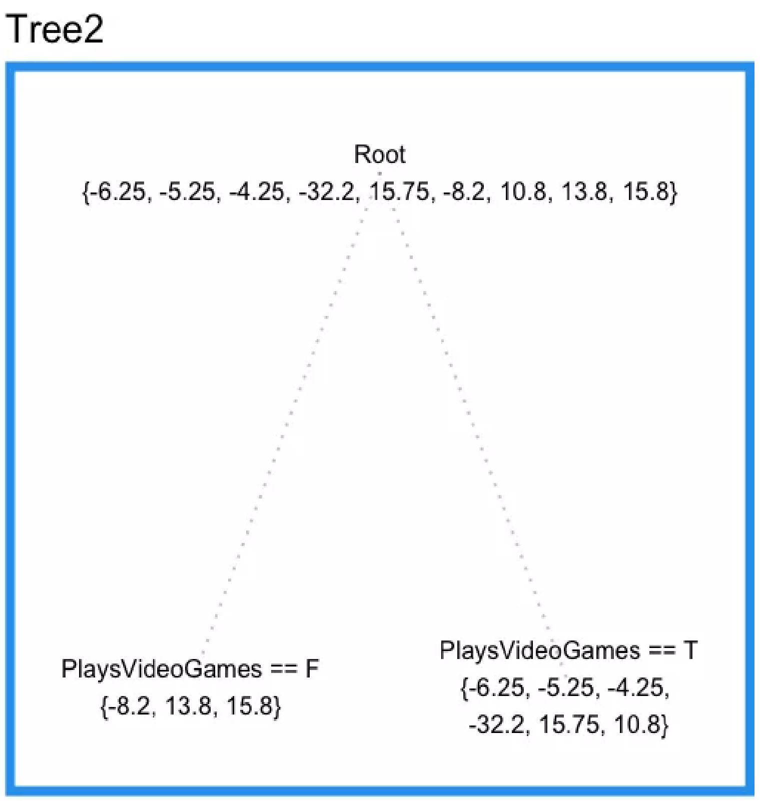
PersonID	Age	LikesGardening	PlaysVideoGames	LikesHats
1	13	FALSE	TRUE	TRUE
2	14	FALSE	TRUE	FALSE
3	15	FALSE	TRUE	FALSE
4	25	TRUE	TRUE	TRUE
5	35	FALSE	TRUE	TRUE
6	49	TRUE	FALSE	FALSE
7	68	TRUE	TRUE	TRUE
8	71	TRUE	FALSE	FALSE
9	73	TRUE	FALSE	TRUE

Now let's model the data with a regression tree. To start, we'll require that terminal nodes have at least three samples. With this in mind, the regression tree will make its first and last split on LikesGardening.



PersonID	Age	Tree1 Prediction	Tree1 Residual
1	13	19.25	-6.25
2	14	19.25	-5.25
3	15	19.25	-4.25
4	25	57.2	-32.2
5	35	19.25	15.75
6	49	57.2	-8.2
7	68	57.2	10.8
8	71	57.2	13.8
9	73	57.2	15.8

Now we can fit a second regression tree to the residuals of the first tree.



Notice that this tree does **not** include *LikesHats* even though **our overfitted regression tree above did**. The reason is because this regression tree is able to consider LikesHats and PlaysVideoGames with respect to all the training samples, contrary to our overfit regression tree which only considered each feature inside a small region of the input space, thus allowing random noise to select *LikesHats* as a splitting feature.

Now we can improve the predictions from our first tree by adding the "error-correcting" predictions from this tree.

PersonID	Age	Tree1 Prediction	Tree1 Residual	Tree2 Prediction	Combined Prediction	Final Residual
1	13	19.25	-6.25	-3.567	15.68	2.683
2	14	19.25	-5.25	-3.567	15.68	1.683
3	15	19.25	-4.25	-3.567	15.68	0.6833
4	25	57.2	-32.2	-3.567	53.63	28.63
5	35	19.25	15.75	-3.567	15.68	-19.32
6	49	57.2	-8.2	7.133	64.33	15.33
7	68	57.2	10.8	-3.567	53.63	-14.37
8	71	57.2	13.8	7.133	64.33	-6.667
9	73	57.2	15.8	7.133	64.33	-8.667

Tree1 SSE	Combined SSE
1994	1765

Gradient Boosting – Draft 1

Inspired by the idea above, we create our first (naive) formalization of gradient boosting. In pseudocode

1. Fit a model to the data, $F_1(x) = y$
2. Fit a model to the residuals, $h_1(x) = y - F_1(x)$
3. Create a new model, $F_2(x) = F_1(x) + h_1(x)$

It's not hard to see how we can generalize this idea by inserting more models that correct the errors of the previous model. Specifically,

$F(x) = F_1(x) \mapsto F_2(x) = F_1(x) + h_1(x) \ldots \mapsto F_M(x) = F_{M-1}(x) + h_{M-1}(x)$
where $F_1(x)$ is an initial model fit to y

Since we initialize the procedure by fitting $F_1(x)$, our task at each step is to find $h_m(x) = y - F_m(x)$.

Stop. Notice something. h_m is just a "model". Nothing in our definition requires it to be a tree-based model. This is one of the broader concepts and advantages to gradient boosting. It's really just a framework for iteratively improving any weak learner. So in theory, a well coded gradient boosting module would allow you to "plug in" various classes of weak learners at your disposal. In practice however, h_m is almost always a tree based learner, so for now it's fine to interpret h_m as a regression tree like the one in our example.

Gradient Boosting – Draft 2

Now we'll tweak our model to conform to most gradient boosting implementations – we'll initialize the model with a single prediction value. Since our task (for now) is to minimize squared error, we'll initialize F with the mean of the training target values.

$$F_0(x) = \underset{\gamma}{\arg\min} \textstyle\sum_{i=1}^n L(y_i, \gamma) = \underset{\gamma}{\arg\min} \textstyle\sum_{i=1}^n (\gamma - y_i)^2 = \displaystyle\frac{1}{n}\sum_{i=1}^n y_i.$$

Then we can define each subsequent F_m recursively, just like before

$F_{m+1}(x) = F_m(x) + h_m(x) = y$, for $m \geq 0$

where h_m comes from a class of base learners $\mathcal{H}$ (e.g. regression trees).

At this point you might be wondering how to select the best value for the model's hyper-parameter m. In other words, how many times should we iterate the residual-correction procedure until we decide upon a final model, F? This is best answered by testing different values of m via cross-validation.