6 DECISION TREES AND GRADIENT BOOSTING

Another important class of algorithms that uses decision trees, besides random forest, is the gradient boosting method.

6.1 Gradient Boosting

We have seen how a single DT can be unstable. Small changes to the training set such as changing of a few sample points or instances and its related features/explanatory variables, or choice of different sub-sets of features (or omitting some) can produce different trees with different prediction on a test sample. A RF on the collection or ensemble of independently trained DTs may "average" the variability of each DT and yield a more accurate forecast. RF also randomizes selection of features - bagging.

Like the random forest (RF) method, the gradient boosting (GB) method is also an ensemble of decision trees (DTs). However, RF and GB differ in the way they combine the decision trees. Random forest combines all independently generated DTs at the end while the gradient boosting method combines the DTs as each subsequent DT is generated in a dependent way from the previous DTs. Each subsequent DT may have a different set of target values, and the DT is also called a boosting tree. The idea of GB is to start with weak trees and then strengthens them as the algorithm proceeds to "solve" for the net errors left from the past DTs.

There are GB methods for continuous or discrete target variable fitting and GB methods for binary classification. We first provide an illustration of the former.

A gradient boosting method (or GB Trees) initially constructs a weak predictor and check for the training error on each sample point. In regression, the initial weak predictor could be just the sample mean $\sum_{j=1}^{N} Y_j/N$ as predictor for every Y_j where Y_j is the target value for the j^{th} training sample point. (Some GB may skip this initial predictor and proceed directly to the first DT.)

Call this initial predictor $F_0 \equiv \bar{Y}$. The initial errors of fitting for each sample point in the training set data are $e_{(0)j} = Y_j - \bar{Y}$, for each j in the training sample.

GBT then builds a first DT, DT(1), to fit/predict $e_{(0)j}$ using features of each sample point j. This mapping or DT works by using each feature value X_{jk}

(each feature k) to decide which next sub-group the sample point j belongs to, until sample point j is led by its features $\{X_{jk}\}$ and by the DT(1) decision rules to a terminal leaf with a fitted/predicted value. The fitting follows the splitting criterion of minimizing weighted mean squared errors in DT. Let the terminal value for a $e_{(0)j}$ be h_{1j}.

The process can be depicted as follows in Figure 6.1.

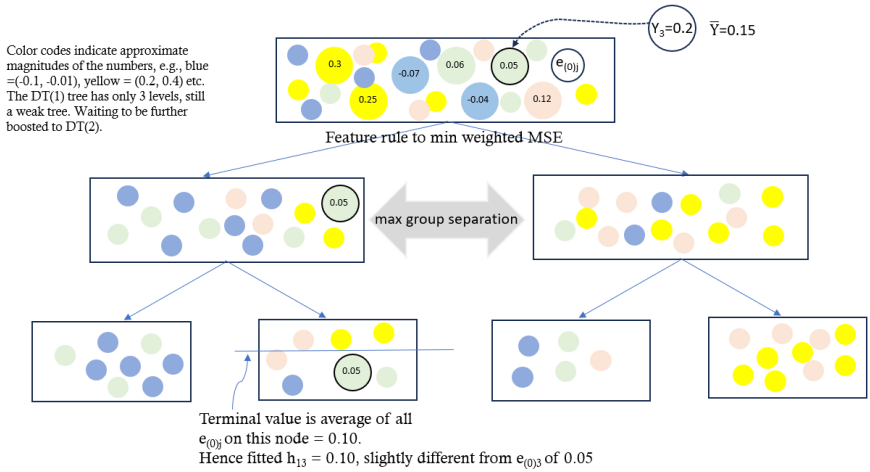


Figure 6.1

In general, we expect to see small differences in targets of $e_{(0)j}$ in DT(1) and the predicted values of h_{1j}. But a good training tree would make this difference close to zero. Even if the DT(1) should in this training data set produce h_{13} exactly equal to $e_{(0)3}$ = 0.05 when the leaf node (final node on tree) is thoroughly resolved to contain only the point or points of the same $e_{(0)j}$ values of 0.05, the DT(1) decision rules when applied to a test sample point with approximately similar features may still yield a slightly different predicted h_{1j} as the features are not identical to that in the training sample point. And if the features are identical, but the test sample point has a slightly different target value than that in the training sample point, then there is still a slight error in prediction.

In GB, to prevent overshooting (rising above the target from below or falling below the target from above), a learning rate (hyperparameter) $\alpha \in (0,1]$, e.g., $\alpha = 0.1$, is imposed on the predicted value h_{1j} within the algorithm.

This regularization does not typically occur in a standalone DT. In this case, the GB algorithm produces the fitted value/predictor of $e_{(0)j}$ as $F_{1j} = \alpha h_{1j}$. The updated value of the prediction of Y_j is $F_0 + F_{1j}$. The fitting/prediction error from DT(1) is then $e_{(1)j} = Y_j - F_0 - F_{1j} = e_{(0)j} - F_{1j}$, for each j in the training sample. Note how the training data set enables use of the label Y_j to determine the next errors $e_{(1)j}$ (for each sample point j in training set) to fit.

Suppose some measure of the total errors $e_{(1)j}$ for all j is not small enough. GBT then builds a second DT in the next stage to train a better DT. And then a third tree to improve on the training, and so on. Thus, unlike the RF method where independent DTs are produced for "averaging", GBT adjusts the algorithm for the next and subsequent DTs (hence dependence of the DTs) to improve on the training based on fitting errors in earlier DTs.

In DT(2), let the terminal (predicted) value for a $e_{(1)j}$ be h_{2j}. The GB algorithm produces the fitted value/predictor of $e_{(1)j}$ as $F_{2j} = \alpha h_{2j}$. The updated value of the prediction of Y_j is $F_0 + F_{1j} + F_{2j}$. The fitting/prediction error from DT(2) is then $e_{(2)j} = Y_j - F_0 - F_{1j} - F_{2j} = e_{(1)j} - F_{2j}$, for each j in the training sample.

Suppose some measure of the total errors $e_{(2)j}$ for all j is still not small enough. The third DT(3) is built to predict $e_{(2)j}$ for all j. The GB algorithm produces the fitted value/predictor of $e_{(2)j}$ as $F_{3j} = \alpha h_{3j}$. The updated value of the prediction of Y_j is $F_0 + F_{1j} + F_{2j} + F_{3j}$. The fitting/prediction error from DT(3) is then $e_{(3)j} = Y_j - F_0 - F_{1j} - F_{2j} - F_{3j} = e_{(2)j} - F_{3j}$, for each j in the training sample. When the maximum number of DTs as in DT(n) is reached or till the measure of errors $e_{(n)j}$, for all j, becomes smaller than a pre-determined size, then the GB tree prediction is $F_0 + F_{1j} + F_{2j} + F_{3j} + \ldots + F_{nj}$ for each Y_j. The boosting regression trees are additive.

Note that in a GB Tree algorithm, the DTs are constructed sequentially in a dependent way whereby previous training errors are considered in building the next tree. Finally, all the trees are added up to form the prediction. The outputs for each of the sequential trees are in a fixed order. This is unlike the RF where each DT can be independently evaluated and hence can be computed in parallel. GB algorithms typically start with weak trees, i.e., trees that produce large errors. Then, stronger trees are subsequently built to predict those errors and add the predicted errors up to form the final prediction. In this sense, subsequent trees put more "weight" to sample points in previous trees with larger errors for subsequent error predictions (since larger errors carry

more "weight" for fitting in squared error sense for optimal splitting). Computationally, GBT is generally more intensive.

We provide a numerical example of the above procedure as follows. Suppose N = 4 and the set of target values, initial predictors (sample mean), and initial fitting errors is shown.

Y_j	0.5	1.0	1.2	1.3
F_0	1.0	1.0	1.0	1.0
$e_{(0)j}$	-0.5	0.0	0.2	0.3

Next, DT(1) computes h_{1j}, hence F_{1j}. Next error is $e_{(1)j} = e_{(0)j} - F_{1j}$.

h_{1j}	-0.8	-0.5	0.4	0.9
F_{1j}	-0.08	-0.05	0.04	0.09
$e_{(1)j}$	-0.42	0.05	0.16	0.21

Next, DT(2) computes h_{2j}, hence F_{2j}. Next error is $e_{(2)j} = e_{(1)j} - F_{2j}$.

h_{2j}	-0.5	0.4	0.7	1.5
F_{2j}	-0.05	0.04	0.07	0.15
$e_{(2)j}$	-0.37	0.01	0.09	0.06

We can see that if the algorithm works, $e(n)_j$ gets smaller toward zero.

More formally, a Gradient Boosting Model develops stronger trees sequentially by minimizing a loss function. The loss function in a GB algorithm has to do with setting up the next target variable (the "error") to fit, in the subsequent boosting trees in a GB approach. It may seem natural just to use subsequent target = "error" = original target less updated fitted/predicted value. But it is also feasible, for example, to use subsequent target = "error" = square of (original target less updated predicted value). It is feasible if eventually the "error" gets toward zero as updated fitted/predicted value closes in on the original target value.

The loss function as in mean squared error (MSE) puts more importance to fit outliers as these impute larger losses. However, if outliers are erroneous, e.g., due to data entry error, then such fitting would produce poor forecasts as the model overfitted outliers. Using mean absolute error (MAE) as loss function would be more robust to effect of outliers in fitting but may not perform well if the outliers reflect genuine higher risk cases.

Suppose the loss function L is MSE or is quadratic in the fitted/predicted "error". $L = \frac{1}{2}\sum_{j=1}^{N}\left(Y_j - \sum_{i=0}^{n} F_{ij}\right)^2$ (for an optimal n number of GB trees) where N is the sample size of the training data set. Typically, n < N. At each boosting stage, e.g., in the algorithm of the mth (m ≤ n) tree, the training loss function would be $\frac{1}{2}\sum_{j=1}^{N}\left(Y_j - F_0 - F_{1j} - \ldots - F_{m-1,j} - F_{mj}\right)^2$. To find the minimum of the loss function, where it is understood that F_{mj} is some function of the features X_{jk}, we could set the derivatives (for each j), i.e., we find features and decision rules on them to train them to match F_{mj} as closely as possible.

$$\frac{\partial}{\partial F_{mj}} \frac{1}{2}\sum_{j=1}^{N}\left(Y_j - F_0 - F_{1j} - \ldots - F_{m-1,j} - F_{mj}\right)^2$$
$$= -\sum_{j=1}^{N}\left(Y_j - F_0 - F_{1j} - \ldots - F_{m-1,j} - F_{mj}\right) = 0.$$

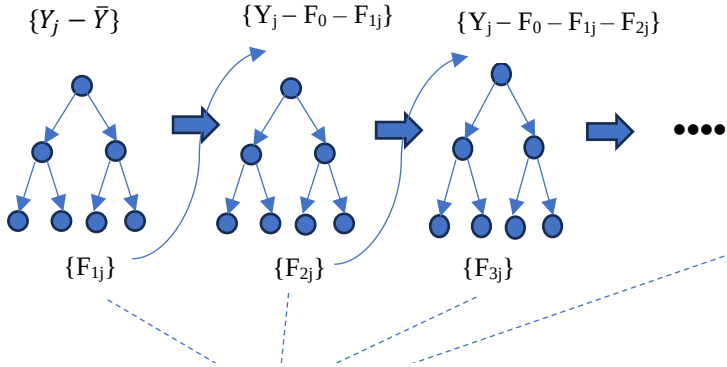
In principle, the minimum loss of zero could be attained when $F_{mj} = Y_j - F_0 - F_{1j} - \ldots - F_{m-1,j} = e_{(m-1)j}$ for each j, or when $\sum_{j=1}^{N} e_{(m-1)j} = \sum_{j=1}^{N} F_{mj}$. These solve the first order conditions. However, these are not possible since F_{mj} is a DT mapping of the features X_{jk}; it is not analytical and exact conditions $F_{mj} = e_{(m-1)j}$ for each j, or $\sum_{j=1}^{N} e_{(m-1)j} = \sum_{j=1}^{N} F_{mj}$ may not be found.

To reduce the loss function toward zero, however, we could train F_{mj} to be as close to or fit/predict given $e_{(m-1)j}$ for each sample point j. This is for m = 1, 2, …, n. The DT terminal value (for each sample point j) is h_{mj}, and the regularized fit/ prediction is $F_{mj} = \alpha h_{mj}$. Note that the "error" $e_{(m-1)j}$ as the target value (for every j) of the mth boosting tree, $e_{(m-1)j} = Y_j - F_0 - F_{1j} - \ldots - F_{m-1,j}$, is also the negative of the gradient term with respect to the loss function. Recall that the gradient is

$$-\left(Y_j - F_0 - F_{1j} - \ldots - F_{m-1,j}\right).$$

Note also that the negative of gradient, $\left(Y_j - F_0 - F_{1j} - \ldots - F_{m-1,j}\right)$, is equal to target less predicted. Hence this approach via reducing "errors" (|gradients|) is updating fit via the "errors" or |gradients|. It is thus called gradient boosting method. Note that at minimum of loss function, gradient is zero, so minimizing gradient is a natural approach.

Visually, GB regression trees for fitting training data can be seen as follows. Note that the features are typically randomly selected for different boosting trees.



GB fit for each j in training data is $F_0 + F_{1j} + F_{2j} + F_{3j} + \ldots + F_{nj}$ for n trees

Figure 6.2

For the test sample or prediction of any test sample point given its features, a DT(1) is illustrated.

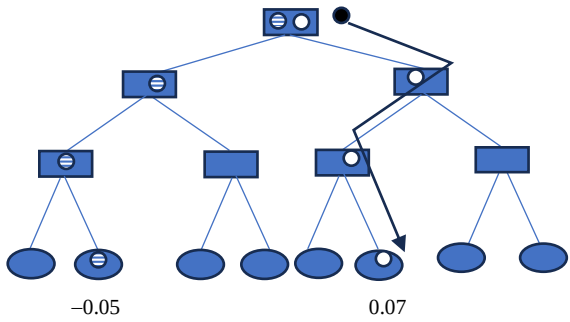
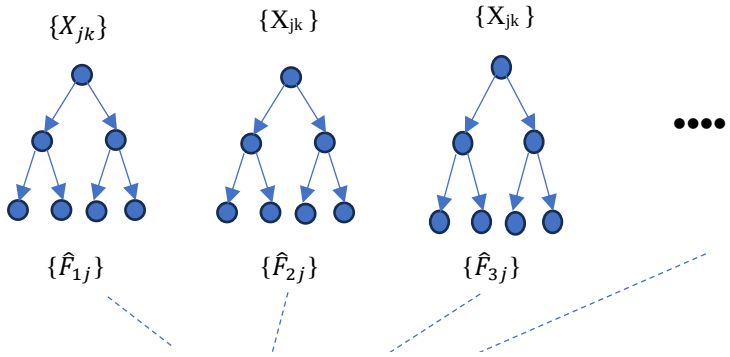


Figure 6.3

The training were done by training data points starting from the root node. The test point (black circle)'s features X_{jk} were used in the DT(1) rules to lead to 0.07.

The trained GB trees DT(1) to DT(n) are used to predict each test sample point j via $\hat{F}_{1j}$ =0.07 from DT(1), $\hat{F}_{2j}$ from DT(2), and so on, without employing the test point's label. Then the GB predictor for the test point j is

$\hat{F}_{0j} + \hat{F}_{1j} + \hat{F}_{2j} + \hat{F}_{3j} + \ldots + \hat{F}_{nj}$. Start term $\hat{F}_{0j}$ could be the same as F_{0j} as it does not involve information from the test point label.



GB prediction for each j in test data is $\hat{F}_{0j} + \hat{F}_{1j} + \hat{F}_{2j} + \hat{F}_{3j} + \ldots + \hat{F}_{nj}$ for n trees

Figure 6.4

The loss function is typically synonymous with the splitting criterion in the case of a single DT or RF. However, in a GB algorithm, the loss function, and the splitting criterion for building decision rules in each of the DTs may be different. In the sklearn GradientBoostingRegressor, the loss function could be squared error L as seen earlier. The splitting criterion can be similarly "squared error" as seen in minimizing weighted mean squared error in RF, but it can also be Friedman mean squared error which is finding the split (optimal feature and its threshold) yielding maximum information gain

$$\frac{w_L w_R}{w_L + w_R}(\bar{Y}^L - \bar{Y}^R)^2$$

where $\bar{Y}^L$ is mean of L training sample points' labels $e_{(m-1)j}$ (for some j) on left node on the mth tree at any level, $\bar{Y}^R$ is mean of R training sample points' labels $e_{(m-1)j*}$ (for some other j*) on right node on the mth tree at the same level, w_L is weight given by $\sum_{j=1}^{L} \hat{p}_j(1-\hat{p}_j)$, w_R is weight given by $\sum_{j*=1}^{R} \hat{p}_{j*}(1-\hat{p}_{j*})$, and $\hat{p}_j$ is probability estimate of label y=1 from a logistic regression of labels of 1 or 0 on the features. The information gain is larger for the same $(\bar{Y}^L - \bar{Y}^R)^2$ when the uncertainties contained in w_L and w_R are larger.

We next provide explanation of the GB method for binary classification. Here instead of fitting sequential errors of continuous variable predictions, we fit sequential residuals that are differences between the observed classification value (1 or 0) and the predicted probability of the case in class 1.

As more training proceeds with sequential or boosting DTs, the probability estimates get more accurate, the residuals (or pseudo residuals – since the probability estimates are model driven and not explicitly observed attributes) reduce toward zero, and eventually the probability estimate for each case j is used to decide if a target belongs to one category or the other. If the case or target has probability > ½, then prediction is that it is in class 1. If the case or target has probability ≤ ½, then prediction is that it is in class 0.

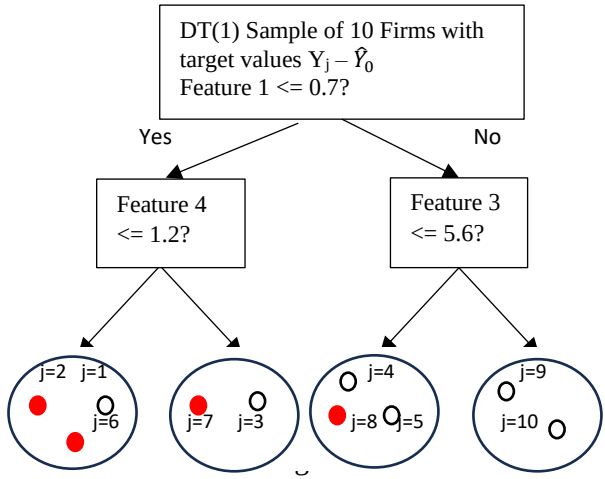
We provide a numerical example of a typical GB Classification algorithm as follows. The targets are credit ratings of firms, belonging to either investment grade (BBB and above) or else speculative grade (BB and below). Suppose there are 10 firms, N = 10. Each firm has features that are accounting variables of the firm. Let the 4 available features be X_{1j}, X_{2j}, X_{3j}, and X_{4j}, where j = 1,2,…,10. Let firm j have target rating of Y_j =1 for investment grade, or else Y_j = 0 for speculative grade. The target values of the firms are shown below.

j	1	2	3	4	5	6	7	8	9	10
Y_j	0	1	0	0	0	1	1	1	0	0

The empirical unconditional probability of Y_j = 1 for each j (supposing that Y_j value is not observed as in a prediction) is 0.4 since 4 out of 10 in the sample have values as 1. A gradient boosting method initially constructs a weak initial predictor of the probability of Y_j = 1 for each Y_j, as 0.4. Note that at the end of the GB algorithm, the final prediction of Y_j is 1 if the final predicted probability is greater than ½, and the final prediction of Y_j is 0 if the final predicted probability is less than or equal to ½.

Call the initial predicted probability p_0, of Y_j =1, to be 0.4. The initial pseudo residuals in the probability fitting of the training data is actual $Y_j - p_0$, either 1 – 0.4 = 0.6 or 0 – 0.4 = – 0.4. The first DT(1) is used to fit/predict $Y_j - p_0$. The DT(1) is shown as follows. Three features were randomly selected to form the decision

rules. (The feature values are just examples.) The splitting criterion is Friedman mean squared error.



In the terminal nodes, the coloured circles represent firms with pseudo residual 0.6 while the uncoloured circles represent firms with pseudo residual – 0.4. The splits are performed based on the Friedman MSE criterion or else weighted mean squared errors criterion on the pseudo residuals. Thus, at the terminal nodes based on the randomly selected features, we obtain some separation of those firms with 0.6 versus those with –0.4. The initial log odds corresponding to $p_0 = \sum_{j=1}^{10} Y_j/10$ is $x_0 = \log[\frac{p_0}{(1-p_0)}]$. The training set also allows identification of j (labels) amongst the circles.

To continue with the next boosting tree DT(2), however, the pseudo residuals have to be updated. This requires specification of a loss function L. A typical loss function for classification problems is the log loss function. Note that the loss criterion is different from the splitting criterion in the GB trees.

$$L = -\sum_{j=1}^{N}\big(Y_j \log p + \big(1 - Y_j\big)\log(1-p)\big) > 0 \qquad (6.1)$$

where p ∈ [0,1] is probability of $Y_j = 1$. There is no loss of generality with any constant multiple on L as these loss functions will have the same solution of p for minimum when L is convex ($\partial^2 L/\partial p^2 > 0$). Log-loss measures "distance" of the predicted probability p to the target values Y_j in the binary classification. Eq. (6.1) can also be expressed as monotonic transform

$$\exp(L) = 1/\left(p^{\sum_{j=1}^{N} Y_j}(1-p)^{\sum_{j=1}^{N}(1-Y_j)}\right) > 1 \qquad (6.2)$$

In Eq. (6.2), higher likelihood function of $p^{\sum_{j=1}^{N} Y_j}(1-p)^{\sum_{j=1}^{N}(1-Y_j)}$ implies that L (loss) is smaller. Lower likelihood function of $p^{\sum_{j=1}^{N} Y_j}(1-p)^{\sum_{j=1}^{N}(1-Y_j)}$ implies that L (loss) is higher. Higher likelihood function and hence smaller loss L implies more accurate prediction of p. Inaccurate predictions are penalized with higher loss values.

We want to find a way to minimize the loss function but using the argument p directly in the function is not helpful as it leads to a final leaf as in a single DT such as via minimizing Gini impurity. We want boosting trees that increments the predicted probability. This can be done using log odds in the argument instead.

If we let x = log (odds), i.e., log p/(1-p), then $p = e^x/(1+e^x)$. Then the log loss function can be written as $L=-\sum_{j=1}^{N}\left(Y_j x - \log(1+e^x)\right)$ in terms of x.

At any node when number of sample points on the node is n ≤ N, i.e., it calculates the loss on that node:

$$L(x) = -\sum_{j=1}^{n}\left(Y_j x - \log(1+e^x)\right) = -\sum_{j=1}^{n} Y_j x + n\log(1+e^x).$$

Using Taylor series expansion around x_0, an initial estimate of the log odds, we have

$$L(x) = L(x_0) + (x-x_0)\left[-\sum_{j=1}^{n} Y_j + \frac{ne^{x_0}}{(1+e^{x_0})}\right] + \frac{1}{2}(x-x_0)^2\left[\frac{ne^{x_0}}{(1+e^{x_0})^2}\right].$$

Higher terms involving $(x-x_0)^3$ and so on are assumed to be small and ignored. This approximation of the log loss function in x allows taking the derivative

$$\left.\frac{dL}{dx}\right|_{x_0} = 0 + \left[-\sum_{j=1}^{n} Y_j + \frac{ne^{x_0}}{(1+e^{x_0})}\right] + (x-x_0)\left[\frac{ne^{x_0}}{(1+e^{x_0})^2}\right] = 0$$

that is set to zero on the right to obtain the optimal x. The second order condition satisfies the case for minimum L. Then,

$$(x - x_0) = \frac{\sum_{j=1}^{n} Y_j - \frac{ne^{x_0}}{(1+e^{x_0})}}{\frac{ne^{x_0}}{(1+e^{x_0})^2}}$$

or, $$x = x_0 + \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)}$$

where $Y_j - p_0$ is the initial pseudo residual. The new value of $x_1 = x$ is meant to reduce the loss function optimally in DT(1). The new updated x_1 increases from x_0 if at the particular terminal node of DT(1), there are more Y_j's =1 than that at the start. It decreases from x_0 if there are less Y_j's =1 than that at the start. Obviously, for any j, a x_{1j} different from x_0 (higher for some j, lower for others) indicates the DT(1) is working well toward separating the types.

The updating involves the gradient dL/dx. However, the algorithm typically puts in a learning rate, e.g., $\alpha = 0.1$, so

$$x_{1j} = x_0 + \alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)}$$

A subscript j is added to x_1 to indicate that now each new log odd may be different for different case or sample point as each may reside in different lower nodes, i.e., in DT(1), $x_{1j} \neq x_{1j*}$. Recall each final splitting produces two terminal nodes with differential pseudo residual values in the two nodes. The $\sum_{j=1}^{n} Y_j$ will be different for each node depending on n points in that node and the sum of Y_j's with target value 1. From this new set of x_{1j}'s for every j in training set , the updated probability estimate is

$$p_{1j} = \frac{e^{x_{1j}}}{1+e^{x_{1j}}}$$

and the updated pseudo residuals are $Y_j - p_{1j}$ for every j. These are then used as labels to train DT(2).

Note that by this stage, the different sample points Y_j may have different corresponding pseudo residuals. Refer to the example in Figure 6.5. The new labels to the same training sample points are shown as follows. Note that $x_0 = \log_e(0.4/0.6) = -0.40547$.

On the left-most leaf in DT(1),

$$\alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)} = 0.1 \times \frac{0.6 + 0.6 - 0.4}{3(0.4)(0.6)} = 0.111111.$$

On the second leaf from the left,

$$\alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)} = 0.1 \times \frac{0.6 - 0.4}{2(0.4)(0.6)} = 0.041667.$$

On the third leaf from the left,

$$\alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)} = 0.1 \times \frac{0.6 - 0.4 - 0.4}{3(0.4)(0.6)} = -0.027778.$$

On the right-most leaf,

$$\alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)} = 0.1 \times \frac{-0.4 - 0.4}{2(0.4)(0.6)} = -0.166667.$$

Hence on the left-most leaf in DT(1), $x_{1j} = x_0 + \alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)}$ = −0.40547 + 0.11111 = − 0.29435. This means every node j = 1,2,6 in that leaf has the same x_{1j} value of − 0.29435. On the second leaf from the left, $x_{1j} = x_0 + \alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)}$ = −0.40547 + 0.041667 = − 0.36380. This means every node j = 3,7 in that leaf has the same x_{1j} value of − 0.36380. On the third leaf from the left, $x_{1j} =$ −0.40547 − 0.02778 = − 0.43324. This means every node j = 4,5,8 in that leaf has the same x_{1j} value of − 0.43324. On the right-most leaf, $x_{1j} =$ −0.40547 − 0.16667 = − 0.57213. This means every node j = 9,10 in that leaf has the same x_{1j} value of − 0.57213.

The corresponding p_{1j}'s and updated pseudo residuals $Y_j - p_{1j}$'s can then be computed for all j. These are shown in the following Table.

Table 6.1

j	Y_j	$\alpha \frac{\sum_{j=1}^{n}(Y_j - p_0)}{np_0(1-p_0)}$	x_{1j}	$p_{1j} = \frac{e^{x_{1j}}}{1+e^{x_{1j}}}$	Updated pseudo residual $Y_j - p_{1j}$	$\Delta p_{1j} = p_{1j} - p_0$
1	0	0.111111	-0.29435	0.426938	-0.42694	0.026938
2	1	0.111111	-0.29435	0.426938	0.573062	0.026938
3	0	0.041667	-0.36380	0.41004	-0.41004	0.01004
4	0	-0.027778	-0.43324	0.393352	-0.39335	-0.00665
5	0	-0.027778	-0.43324	0.393352	-0.39335	-0.00665
6	1	0.111111	-0.29435	0.426938	0.573062	0.026938
7	1	0.041667	-0.36380	0.41004	0.58996	0.01004
8	1	-0.027778	-0.43324	0.393352	0.606648	-0.00665
9	0	-0.166667	-0.57213	0.360745	-0.36075	-0.03925
10	0	-0.166667	-0.57213	0.360745	-0.36075	-0.03925

Next the second boosting tree, DT(2), is built to fit/predict the new pseudo residuals or targets $Y_j - p_{1j}$. Nodes or leaves (not just end leaves) have updated log-odds as follows.

$$x_{2j} = x_{1j} + \alpha \frac{\sum_{j=1}^{n}(Y_j - p_{1j})}{\sum_{j=1}^{n} p_{1j}(1-p_{1j})}$$

for j's within each node. Basically at each m^{th} boosting tree, x_{mj} is updated. This leads to new fitted p_{mj}, new updated pseudo residual $Y_j - p_{mj}$, next tree fitting and so on.

The algorithm then continues in the same way until there is no significant improvement in the log odds, i.e., x_{nj}. At the end of n trees, i.e., at end of DT(n), the probability estimates p_{nj} for each j are fitted and the GB trees DT(1) up to DT(n) can now be used to predict p_{nk} for each test sample point k.

An alternative way to think about it is as follows. At the end of n trees, i.e., at end of DT(n), the probability estimates p_{nj} for each j are computed as p_{nj} =

$\Delta p_{nj} + \Delta p_{n-1,j} + \ldots + \Delta p_{3j} + \Delta p_{2j} + \Delta p_{1j} + p_0$. If $p_{nj} > ½$, then that j^{th} case is predicted as $Y_j = 1$. Otherwise, it will be predicted as $Y_j = 0$. The number of boosting trees n can be large, e.g., 100, especially if the learning rate α is kept low. It is common for an effective GB algorithm in a well-defined data set with predictive patterns to enable a 100% or close to 100% training fit, i.e., 100% accuracy in training.

Visually, GB classification trees for fitting training data can be seen as follows.

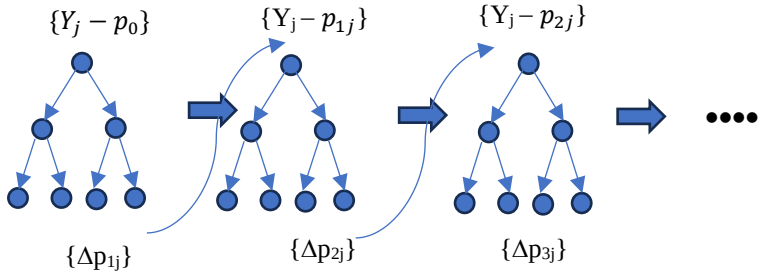


Figure 6.6

Note that the features are typically randomly selected for different boosting trees. GB probability predictor for each j is $p_0 + \Delta p_{1j} + \Delta p_{2j} + \Delta p_{3j} + \ldots\ldots + \Delta p_{nj}$.

For the test sample or prediction of any sample point j given its features, the trained GB classification trees DT(1) to DT(n) are used to predict Δp_{1j} from DT(1), Δp_{2j} from DT(2), and so on. Then the GB probability predictor for each j is $p_0 + \Delta p_{1j} + \Delta p_{2j} + \ldots + \Delta p_{nj}$. If $p_{nj} > ½$, then that j^{th} case is predicted as $Y_j = 1$. Otherwise, it will be predicted as $Y_j = 0$.

There are several points worth noting. Firstly, a different loss function such as the mean squared error may not work as well here as $L(x) = \frac{1}{N}\sum_{j=1}^{N}(Y_j - p)^2 = \frac{1}{N}\sum_{i=j}^{N}\left(Y_j - \frac{e^x}{1+e^x}\right)^2$ is non-convex given Y_j's for some x. Non-convexity does not allow a global minimum to be found and local minimum may not lead to effective predictors. If we try to find the minimum in $\frac{1}{N}\sum_{j=1}^{N}(Y_j - p)^2$ by taking p as the control variable, then its solution is just

$\hat{p} = \frac{1}{N}\sum_{j=1}^{N} Y_j$ which is just the proportion of Y_j taking the value 1. $\hat{p}$ is also the initial predictor p_0 in the above example. It is the same for all sample points j, so it is not helpful to predict

Secondly, if in the terminal node of each tree we had used updated estimate of p directly as the empirical probability on that node, e.g., $p_1 = 2/3$ (based on j=1,2,6 on the left-most leaf) without going through the gradient involving log odds, then the DT becomes an ordinary DT, replacing the gini index with probability residual, without boosting as the levels on the DT can continue to grow without the need to update the targets.

Thirdly, unlike RF where bagging uses bootstrapping – random sampling with replacements, GB may use random sampling of a smaller set of sample points for its targets but without replacements (i.e., no repeated sample points) for the boosting trees. The smaller set enables faster computations. This alternative is sometimes called stochastic gradient boosting.

Fourthly, many different varieties of GB based on different loss functions, different split criteria, and different ways of combining the boosting trees, can be built. There are some general comparisons but specifically how one ensemble tree algorithm will perform relative to another depends also on the data.

Fifthly, GB trees may require more tuning of the hyperparameter(s) – setting learning rate, pruning some nodes, setting maximum number of levels per tree, etc.

An example of a popular alternative to the Gradient Boosting algorithm is the LightGBM (light gradient-boosting machine). LightGBM (from a different original builder and is not part of sklearn) generally has faster speed and places less burden on computer memory. In LightGBM (which can apply to both RF and boosting method), each tree grows not level-wise as in traditional trees, but leaf-wise. Instead of extending the levels down with two nodes at the next level, then 4 nodes, etc., the nodes are only extended down from the previous node with the largest information gain or minimum loss. In principle, if we can grow the trees as long as possible, the lightGBM should produce similar prediction result with the regular GB method. However, with regularizations such as limiting maximum depth of trees, the lightGBM can perform marginally better. The following diagram shows how one lightGBM tree is constructed in the training data.

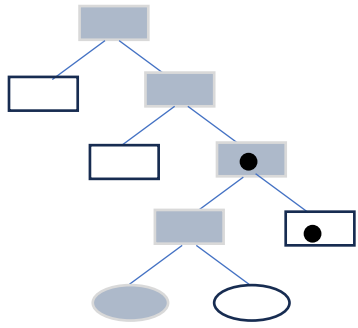


Figure 6.7

Shaded decision nodes and final leaf indicate the path with maximum Gini impurity decrease or maximum decrease in mean square error, depending on whether it is a classification or an estimation tree. The tree does not grow in the non-shaded nodes. Different versions of lightGBM may use different methods of splitting. When a sample point (black) is "stuck" in an upper node – that node's score/values become its predicted value. Score refers to regression estimate using mean of sample labels in node, and other information. Values refer to numbers of the types and can be used to predict probability in a RF for binary classification.

6.2 Worked Example GB – Data

We continue with the credit rating accounting data set corporate_rating2.csv used in chapter 5 for the DT and RF algorithms. The program file is Chapter6-1.ipynb.

Sklearn GradientBoostingClassifier is used with specification of 500 DTs (n_estimators = 500). We can set random selection of features, e.g., max_features = 'sqrt' which means that each DT randomly uses $\sqrt{25} = 5$ features of the 25 total in constructing the branching in each of the 500 DTs. However, because of the learning to solve for the net error in subsequent DTs, there is a learning rate just as in Gradient Descent. Here we fine-tune the learning rate to attain a higher accuracy or lower loss function. See code line [14].

Gradient Boosting

In [14]:
```
from sklearn.ensemble import GradientBoostingClassifier

### updated version showing all the options
### class sklearn.ensemble.GradientBoostingClassifier(*, loss='log_loss', learning_rate=0.1, n_estimators=100,
### subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
### max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0,
### max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)

GB_model = GradientBoostingClassifier(n_estimators=500, random_state=1, max_features="sqrt", learning_rate=0.01, max_depth=24)
GB_model.fit(X_train, y_train)

y_pred_GB = GB_model.predict(X_test)
Accuracy_GB = metrics.accuracy_score(y_test, y_pred_GB)
print("GB Accuracy:",Accuracy_GB)
```

```
GB Accuracy: 0.8051181102362205
```

The confusion matrix and the classification report for the test are shown in code lines [15] and [16].

```
In [15]: from sklearn.metrics import confusion_matrix
         confusion_matrix = confusion_matrix(y_test, y_pred_GB)
         print(confusion_matrix)
```

```
[[151  64]
 [ 35 258]]
```

```
In [16]: from sklearn.metrics import classification_report
         print(classification_report(y_test, y_pred_GB))
```

```
              precision    recall  f1-score   support

           0       0.81      0.70      0.75       215
           1       0.80      0.88      0.84       293

    accuracy                           0.81       508
   macro avg       0.81      0.79      0.80       508
weighted avg       0.81      0.81      0.80       508
```

In this investment grade/speculative grade prediction problem, the accuracy of the GB here is 80.51%. The AUC is 89.62%. The prediction performance is marginally better than RF with accuracy of 79.53% and AUC of 88.69%.

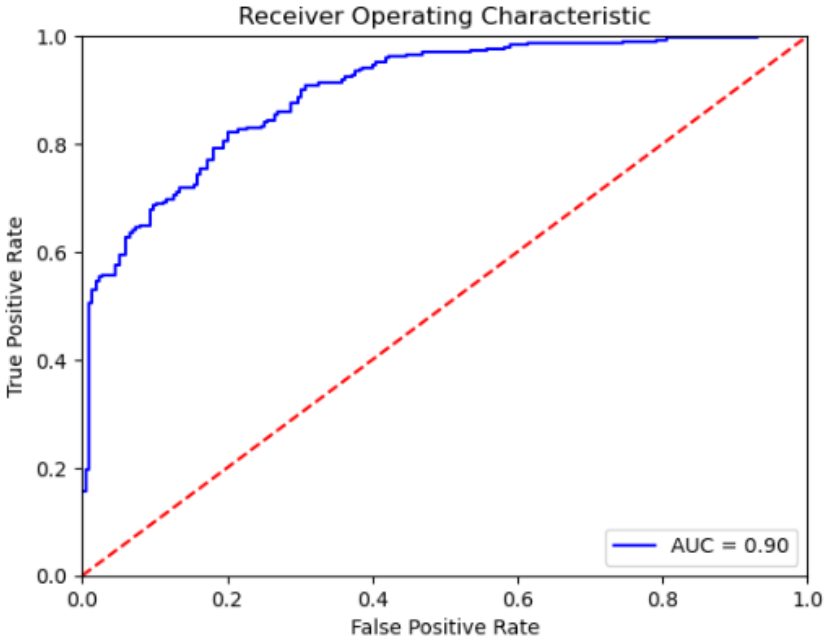
The ROC curve is shown below after computing the various True positive rates and False positive rates based on different threshold probabilities.

```
In [17]: import sklearn.metrics as metrics
         # calculate the fpr and tpr for all thresholds of the classification
         preds_GB = GB_model.predict_proba(X_test)[:,1]

         fpr, tpr, thresholds = metrics.roc_curve(y_test, preds_GB)
             ### matches y_test of 1's and 0's versus pred prob of 1's for each of the 508 test cases
             ###    sklearn.metrics.roc_curve(y_true, y_score,...) requires y_true as 0,1 input and y_score as prob inputs
             ###    this metrics.roc_curve returns fpr, tpr, thresholds (Decreasing thresholds used to compute fpr and tpr)
         roc_auc_GB = metrics.auc(fpr, tpr)
             ### sklearn.metrics.auc(fpr,tpr) returns AUC using trapezoidal rule
         roc_auc_GB
```

```
Out[17]: 0.8961822366854512
```

```
In [18]: import matplotlib.pyplot as plt
         plt.title('Receiver Operating Characteristic')
         plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc_GB)
         plt.legend(loc = 'lower right')
         plt.plot([0, 1], [0, 1],'r--')
         plt.xlim([0, 1])
         plt.ylim([0, 1])
         plt.ylabel('True Positive Rate')
         plt.xlabel('False Positive Rate')
         plt.show()
```



For comparison, we use the sklearn LogisticRegression for prediction. The results are shown below. It is seen that RF and GB perform significantly better than the logit regression – indicating the power of ensemble methods when there is a large number of features in a complex manner of data patterns.

In [19]:
```
# Logistic Regression
from sklearn.linear_model import LogisticRegression

LR = LogisticRegression(random_state=1,
                        multi_class='multinomial',
                        solver='newton-cg')
LR = LR.fit(X_train, y_train)

y_pred_LR = LR.predict(X_test)
Accuracy_LR = metrics.accuracy_score(y_test, y_pred_LR)
print("LR Accuracy:",Accuracy_LR)
```

```
LR Accuracy: 0.6476377952755905
```

```
In [20]: from sklearn.metrics import confusion_matrix
         confusion_matrix = confusion_matrix(y_test, y_pred_LR)
         print(confusion_matrix)
```

```
[[ 70 145]
 [ 34 259]]
```

```
In [21]: from sklearn.metrics import classification_report
         print(classification_report(y_test, y_pred_LR))
```

```
              precision    recall  f1-score   support

           0       0.67      0.33      0.44       215
           1       0.64      0.88      0.74       293

    accuracy                           0.65       508
   macro avg       0.66      0.60      0.59       508
weighted avg       0.65      0.65      0.61       508
```

```
In [22]: import sklearn.metrics as metrics
         # calculate the fpr and tpr for all thresholds of the classification
         preds_LR = LR.predict_proba(X_test)[:,1]

         fpr, tpr, thresholds = metrics.roc_curve(y_test, preds_LR)
             ### matches y_test of 1's and 0's versus pred prob of 1's for each of the 508 test cases
             ###    sklearn.metrics.roc_curve(y_true, y_score,...) requires y_true as 0,1 input and y_score as prob inputs
             ###    this metrics.roc_curve returns fpr, tpr, thresholds (Decreasing thresholds used to compute fpr and tpr)
         roc_auc_LR = metrics.auc(fpr, tpr)
             ### sklearn.metrics.auc(fpr,tpr) returns AUC using trapezoidal rule
         roc_auc_LR
```

```
Out[22]: 0.6683705055956821
```

We also apply LightGBM. We have to install the lightgbm program from a different source. The accuracy of lightGBM is 83.27%.

```
In [27]: !pip3 install lightgbm
```

```
In [28]: import lightgbm as lgb
         from lightgbm.sklearn import LGBMClassifier
```

```
In [29]: # Creating an object for model and fitting it on training data set
         model = LGBMClassifier(learning_rate=0.1)
         model.fit(X_train, y_train)

         # Predicting the Target variable
         predLGBM = model.predict(X_test)
         print(predLGBM)
         accuracy = model.score(X_test, y_test)
         print(accuracy)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1 1 0 1 1 1 1 1 0 0 1 0 0 1 1 0 1
 0 1 1 1 1 0 0 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 1 1 1 0 1 1
 0 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 0 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1
 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 1 1 0 0 0 1 1 0 1 1 1 1 1 1 1
 1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1
 1 1 1 0 0 1 1 0 1 1 1 1 0 1 1 0 0 1 1 1 0 0 1 1 1 0 1 1 0 0 0 1 1 0 0 0 0
 0 0 1 1 1 1 1 0 1 1 0 1 1 0 0 1 0 0 1 1 1 0 0 1 1 0 0 1 1 0 0 1 0 0
 1 0 1 1 1 0 0 0 1 0 1 1 0 1 0 1 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 1 1 0 0 1 1
 0 1 1 1 1 0 0 0 1 0 1 0 0 1 0 0 1 1 0 1 0 1 0 0 1 1 1 1 0 1 1 1 0 1 1 1 1
 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 1 0 0 0
 0 1 1 1 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 0 1 0 0
 1 1 0 0 1 1 1 1 0 1 1 1 1 0 0 1 1 1 1 1 1 1 0 1 0 0 1 0 1 0 1 1 0 1 1 0 1 1
 1 1 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 0 1 0 1 1 0 1 1 1 0 0]
0.8326771653543307
```

The matrix of 1's and 0's shows the prediction of the 508 test cases. The confusion matrix and classification report of lightGBM are shown in code lines [30] and [31].

```
In [30]: from sklearn.metrics import confusion_matrix
         confusion_matrix = confusion_matrix(y_test, predLGBM)
         print(confusion_matrix)
```

```
[[160  55]
 [ 30 263]]
```

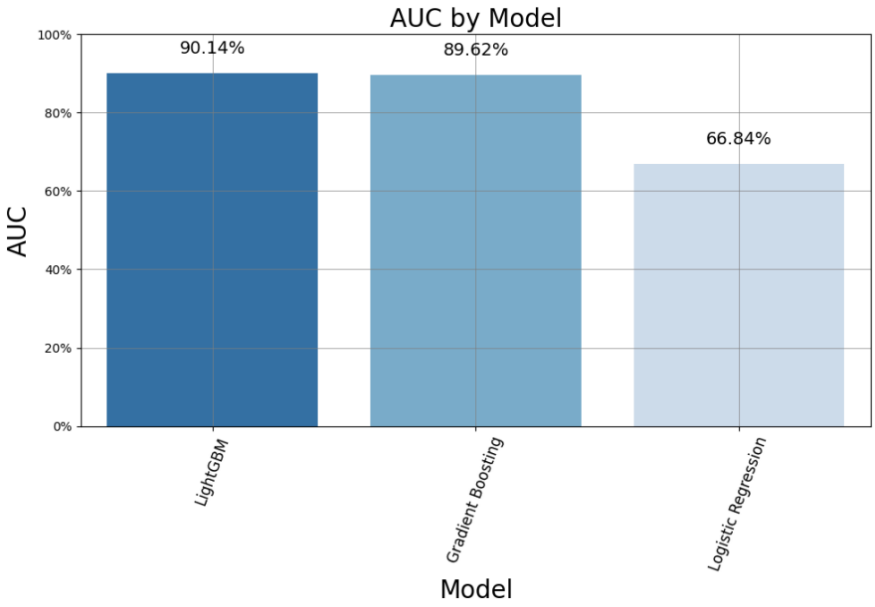
```
In [31]: from sklearn.metrics import classification_report
         print(classification_report(y_test, predLGBM))
```

```
              precision    recall  f1-score   support

           0       0.84      0.74      0.79       215
           1       0.83      0.90      0.86       293

    accuracy                           0.83       508
   macro avg       0.83      0.82      0.83       508
weighted avg       0.83      0.83      0.83       508
```

The comparison of AUC performances of the Gradient Boosting method, the logistic regression, and the LightGBM method are shown as follows.



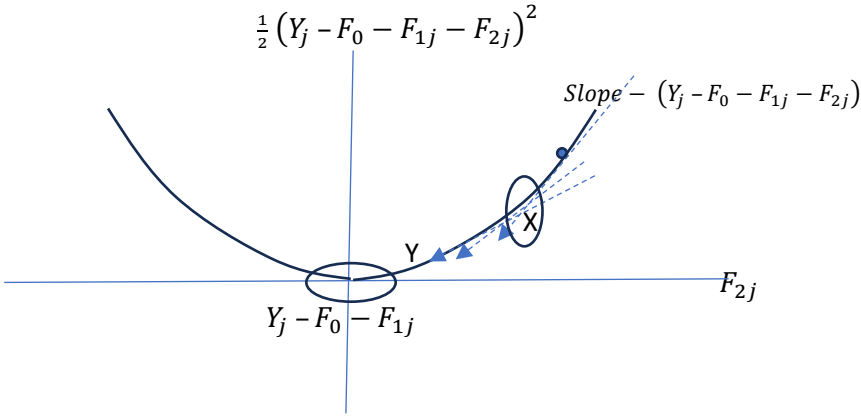
6.3 Summary

In summary, ensemble methods include the bagging approach of RF and the gradient boosting approach of GB, LightGBM, and others. Each DT in a RF of say 500 trees produces a prediction x_i. The prediction of the trees used the same decision rules formulated in the training sample for each of the ensemble of trees. Thus, it is possible that the formulated rules could produce a biased estimate and hence biased prediction. For regression trees, the RF basically yields an averaged prediction from the 500 trees. The variance of this average is $var\left(\frac{1}{500}\sum_{i=1}^{500} x_i\right) \approx \frac{1}{500}\, var(x_i)$ which is much smaller than the variability of an individual DT prediction, assuming the predictions from different trees are approximately independent. Hence bagging decreases variance in the prediction but may not decrease the bias if there is any.

For gradient boosting, the idea is to use gradient to boost subsequent trees to reduce the loss function toward zero. Typically, if the loss function gets to near-zero, the prediction would be very accurate. Hence the gradient boosting

approach reduces bias in trees toward zero, but the variability may be larger than in RF.

The above idea can be illustrated as follows in the regression tree for continuous or discrete target variable fitting. Suppose the loss function L = $\frac{1}{2}\sum_{j=1}^{N}\left(Y_j - \sum_{i=0}^{n} F_{ij}\right)^2$ for n trees is MSE or is quadratic in the fitted/predicted "error". Suppose n = 2.



In RF, suppose most re-sampled test data trees based on the trained decision rules predict with a bias and end up close to point X (circled region). The averaged prediction of the RF ensemble would also in general be biased but with less variance. The GB trees however would sequentially improve by training on the gradients toward $Y_j - F_0 - F_{1j}$ so that the prediction would have less bias, but perhaps more variance (circled region). Note F_{1j} was also trained using gradient adjustment.

In GB, if the learning rate is too small, the algorithm will require many subsequent trees (iterations) to get close to the minimum. But if the learning rate is too high, the prediction may overshoot the target and end up having the next iteration or tree to revert the other direction. Thus, the learning may take longer.

XGBoost is GB method with added regularizations on the optimization of the loss function. With adequate hyperparameter tuning, it can produce slightly better predictive performance than the GBM. AdaBoost is another boosting method that is popular.

Besides the popular bagging via RF and the boosting gradient models in ensemble prediction, another ensemble approach is stacking. Stacking uses different models, e.g., RF, GB, SVM, NN to perform predictions. The different model predictions (from the different machine learners), called the intermediate predictions, are then possibly averaged to obtain the final ensemble prediction. This final prediction is said to be stacked on top of the intermediate models.

References

Aurelien Geron, (2019), "Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow", 2nd ed., O'Reilly Publisher.
Marcos Lopez De Prado (2018), "Advances in Financial Machine Learning", Wiley.
Jerome H. Friedman, (2002), "Stochastic Gradient Boosting", Computational Statistics & Data Analysis, Vol.38 (4), 367—378.