QF634 APPLIED QUANTITATIVE RESEARCH METHODS
LECTURE 6

Lecturer: Prof Emeritus Lim Kian Guan

Gradient Boosting, Shap Values

Gradient Boosting

- A single DT can be unstable. 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). 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.

Gradient Boosting

- 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.)

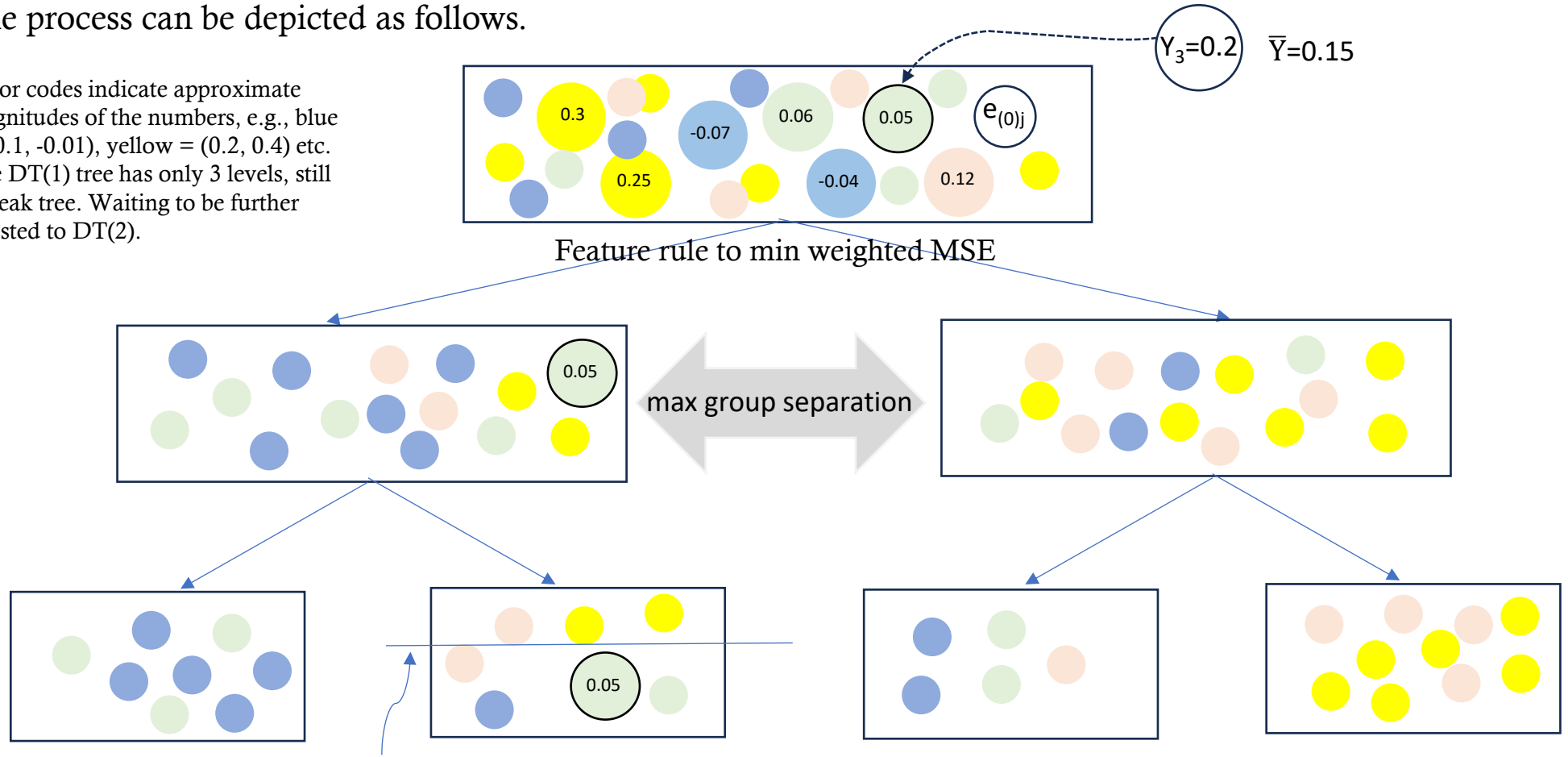
- Call this initial predictor $F_0 \equiv \bar{Y}$. The initial errors of fitting for each sample point in the training set data are $e_{(0)j} = Y_j - \bar{Y}$, for each j in the training sample.

- 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}.

Gradient Boosting

The process can be depicted as follows.

Color codes indicate approximate magnitudes of the numbers, e.g., blue =(-0.1, -0.01), yellow = (0.2, 0.4) etc. The DT(1) tree has only 3 levels, still a weak tree. Waiting to be further boosted to DT(2).



Terminal value is average of all $e_{(0)j}$ on this node = 0.10.
Hence fitted h_{13} = 0.10, slightly different from $e_{(0)3}$ of 0.05

Gradient Boosting

- 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.
- 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 $F_0 + F_{1j}$. Note Y_j is still in the training data.
- 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.
- 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.

Gradient Boosting

- 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.

- In GB, 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.

Gradient Boosting

- We provide a numerical example. Suppose training sample size 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, $e_j(n)$ gets smaller toward zero.

Gradient Boosting

- More formally, a Gradient Boosting Model develops stronger trees sequentially by minimizing a loss function. This loss function is typically synonymous in the case of RF, but has an added purpose in a GB algorithm, and may be different from the splitting criterion, as for example, in the sklearn GB Classifier for binary classifications.

- The loss function in a GB algorithm is directly connected with setting up the next target variable (the "error") to fit, in the subsequent boosting trees in a GB approach. We just saw how the subsequent target = "error" = original target less updated fitted/predicted value, viz., $e_{(1)j} = Y_j - (F_0 + F_{1j})$, $e_{(2)j} = Y_j - (F_0 + F_{1j} + F_{2j})$, $e_{(3)j} = Y_j - (F_0 + F_{1j} + F_{2j} + F_{3j})$, and so on.

- Subsequent trees put more "weight" or consideration to sample points in previous trees with larger errors such as in squared error sense. 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.

Gradient Boosting

- 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.

At each boosting stage, e.g., in the algorithm of the m^{th} ($m \leq n$) tree, the training loss function would be $\frac{1}{2}\sum_{j=1}^{N}\left(Y_j - F_0 - F_{1j} - \ldots - F_{m-2,j} - F_{m-1,j} \right)^2$. To find the minimum of the loss function, supposing $F_{m-1,j}$ is some function of the features X_{jk}, we could set the following derivatives to zeros, i.e.,

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

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

- In principle, the minimum loss of zero could be attained when $\sum_{j=1}^{N} F_{m-1,j} = \sum_{j=1}^{N} Y_j - F_0 - F_{1j} - \ldots - F_{m-2,j}$. However, as we are dealing with DTs, it is not possible to explicitly derive an analytical function of $F_{m-1,j}$ in terms of X_{jk} for every j such that $\sum_{j=1}^{N} F_{m-1,j} = \sum_{j=1}^{N} Y_j - F_0 - F_{1j} - \ldots - F_{m-2,j}$. But we can try to find for each j, $F_{m-1,j}$ that is close to $Y_j - F_0 - F_{1j} - \ldots - F_{m-2,j} = e_{(m-2)j}$ using the DT algorithm. And we can try to find for each j, $F_{m,j}$ that is close to $Y_j - F_0 - F_{1j} - \ldots - F_{m-2,j} - F_{m-1,j} = e_{(m-1)j}$ and so on.

Gradient Boosting

- Note that the "error" $e_{(m-1)j}$ as the target value (for every j) of the m[th] boosting tree, $e_{(m-1)j} = Y_j - F_0 - F_{1j} - \ ... - F_{m-1,j}$, is also element in the negative of the gradient term $\frac{\partial L}{\partial F_{m-1,j}}$

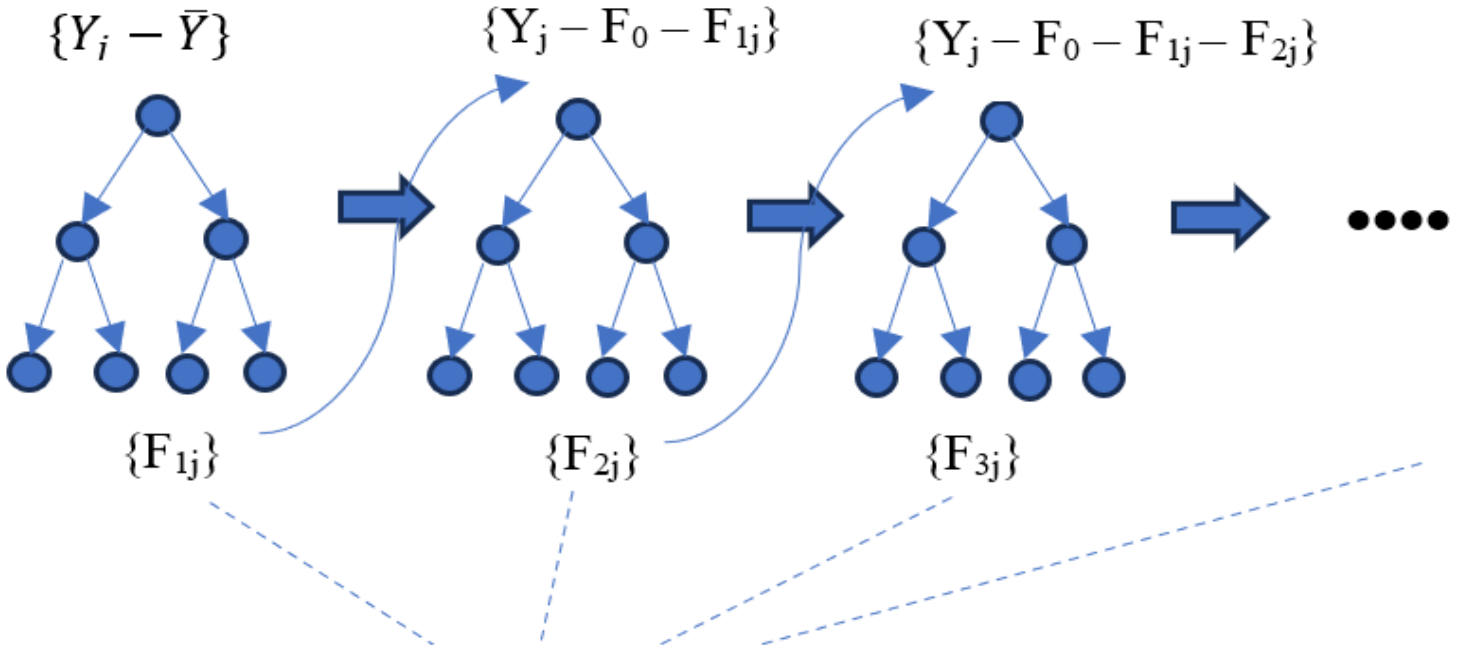
$$- \sum_{j=1}^{N} \left(Y_j - F_0 - F_{1j} - ... - F_{m-2,j} - F_{m-1,j} \right)$$

with respect to the loss function. Hence this approach via reducing "errors" or reducing the gradients toward zeros is thus called gradient boosting method.

Reducing the loss function is directly connected with setting up the next target errors $e_{(m-1)j}$ for fitting.

Gradient Boosting

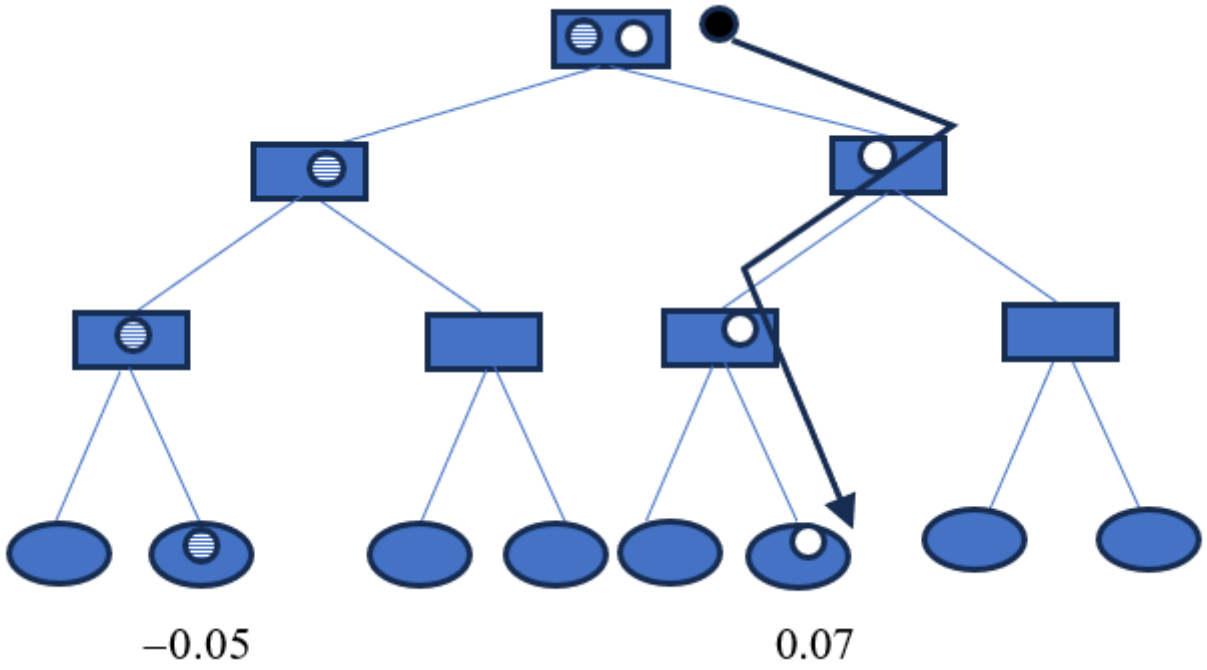
- Visually, GB regression trees for fitting training data can be seen as follows. Note that the features are typically randomly selected for different boosting trees. The quantities on the top line $\{Y_j - \bar{Y}\}$, $\{Y_j - F_0 - F_{1j}\}$, $\{Y_j - F_0 - F_{1j} - F_{2j}\}$, etc., denote the target errors $e_{(0)j}$, $e_{(1)j}$, $e_{(2)j}$ for the trees DT(1), DT(2), DT(3), and so on. The bottom line quantities $\{F_{1j}\}$, $\{F_{2j}\}$, $\{F_{3j}\}$, etc. denote the predicted errors of the DTs.



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

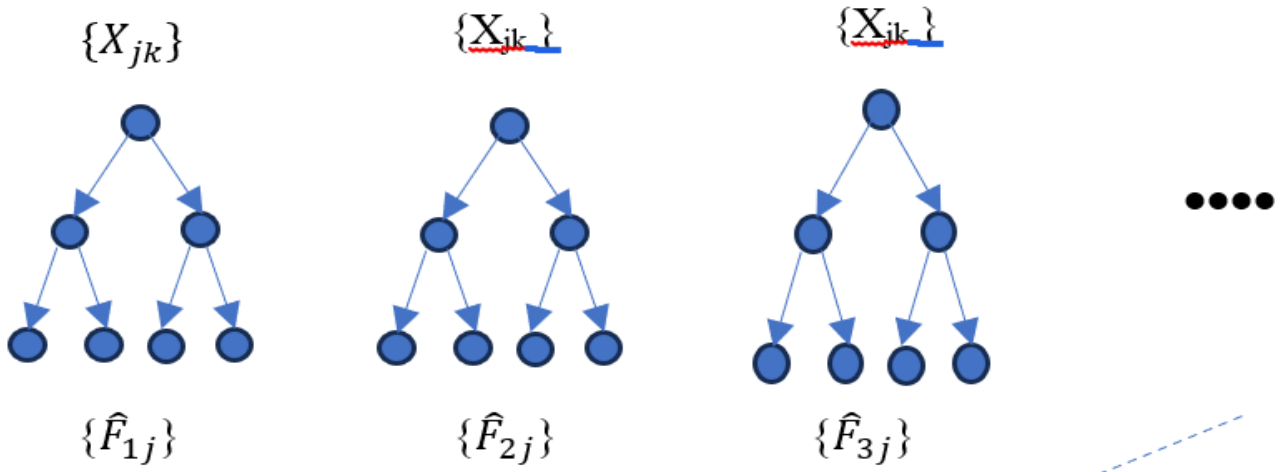
Gradient Boosting

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



Gradient Boosting

• 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

Gradient Boosting

In GB method for binary classification, 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.

Gradient Boosting

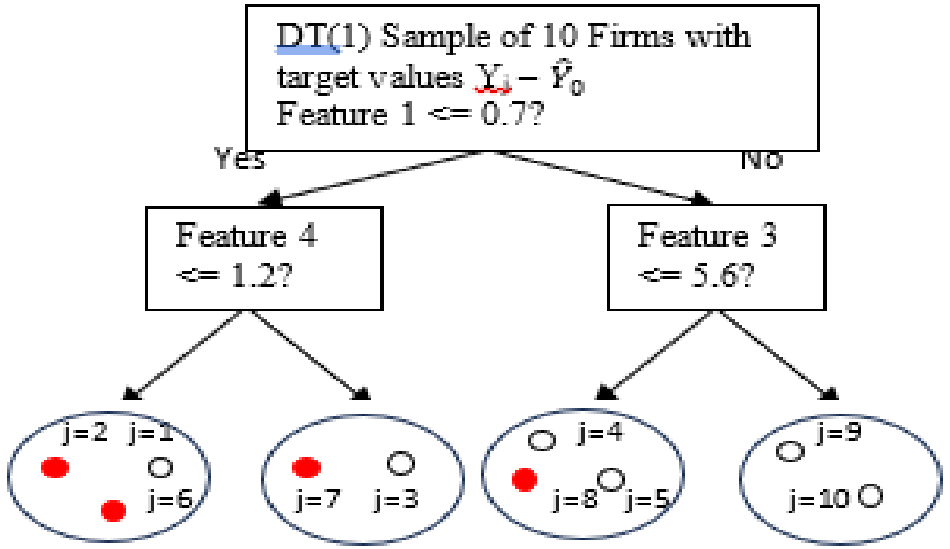
- Numerical example of a typical GB Classification algorithm as follows. The targets are credit ratings of firms, belonging to either investment grade (BBB and above) or else speculative grade (BB and below). Suppose there are 10 firms, N = 10. Each firm has features that are accounting variables of 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 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 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.

Gradient Boosting

- 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.)



- 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 or else weighted mean squared errors 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 $x_0 = \log[\frac{p_0}{(1-p_0)}]$.

Gradient Boosting

- 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. Note that the loss criterion is different from the splitting criterion in the GB trees. A typical loss function for classification problems is the log loss function. We employ L without the multiple of 2. There is no loss of generality with any constant multiple on L as L will have the same minimum point when L is convex ($\partial^2 L/\partial p^2 > 0$)

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

  where p $\in$ [0,1] is probability of $Y_j = 1$.

- 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.

Gradient Boosting

- We want to find a way to minimize the loss function such that the boosting trees could increment the predicted probability. This can be done using log odds in the argument.
- 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,

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

- 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 higher derivatives of $\frac{ne^x}{(1+e^x)^2}$ and so on 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.

Gradient Boosting

- 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 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), $x_{1j} \neq x_{1j^*}$ for different instances j and j*. 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).

Gradient Boosting

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 $x_0 = \log_e(0.4/0.6) = -0.4055$. On the left-most leaf in DT(1),

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

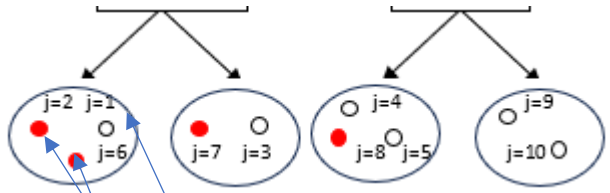
On the second leaf from the left,

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

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



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

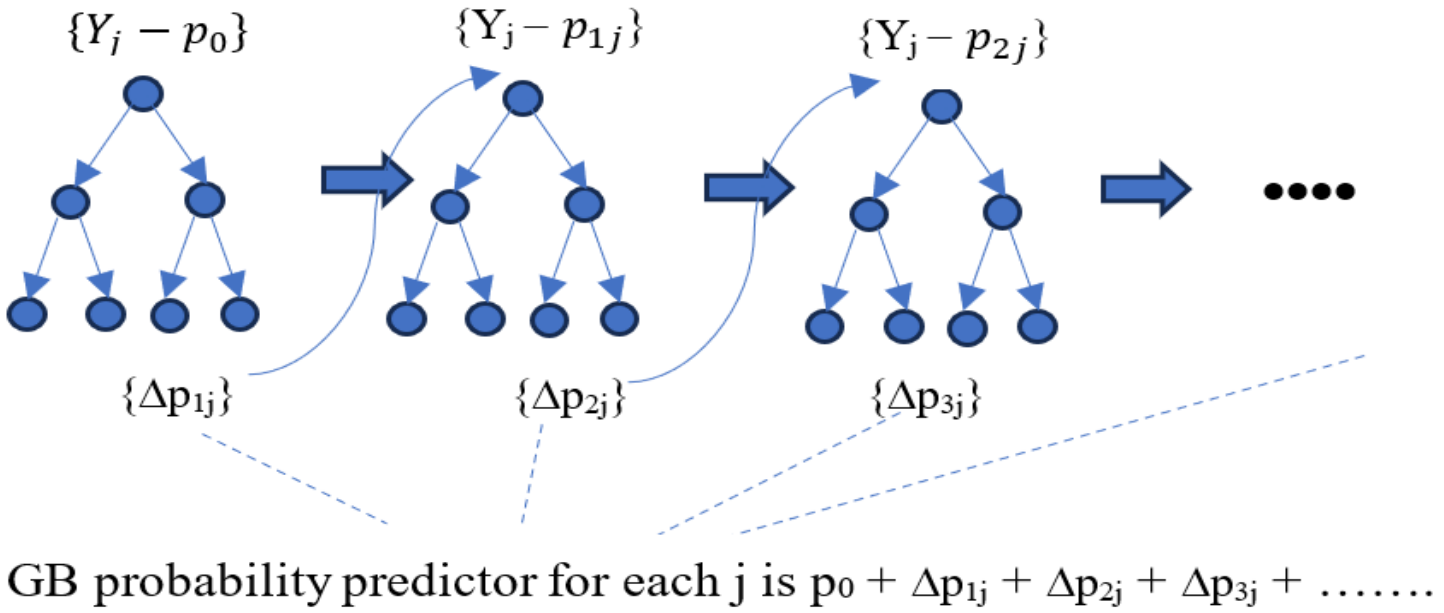
Gradient Boosting

- Next the second boosting tree, DT(2), is built to fit/predict the new pseudo residuals or targets $Y_j - p_{1j}$. The algorithm then continues in the same way until there is no significant improvement in the log odds, i.e., x.
- 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.
- 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 $p_{nj} = \Delta p_{nj} + \Delta p_{n-1,j} + \ldots\ldots + \Delta p_{3j} + \Delta p_{2j} + \Delta p_{1j} + p_0$. 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.

Gradient Boosting

- Visually, GB method for classification can be seen as follows. Note that the features are typically randomly selected for different boosting trees.



- 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 $Y_j = 0$.

Gradient Boosting

- 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) = $\frac{1}{N}\sum_{j=1}^{N}\left(Y_j - p\right)^2 = \frac{1}{N}\sum_{i=j}^{N}\left(Y_j - \frac{e^x}{1+e^x}\right)^2$ is non-convex in x given Y_j's. Non-convexity does not allow a global minimum to be found and local minimum may not lead to effective predictors.
- 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.

Light Gradient Boosting Machine (Light GBM)

- 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.
- 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.

Light Gradient Boosting Machine (Light GBM)

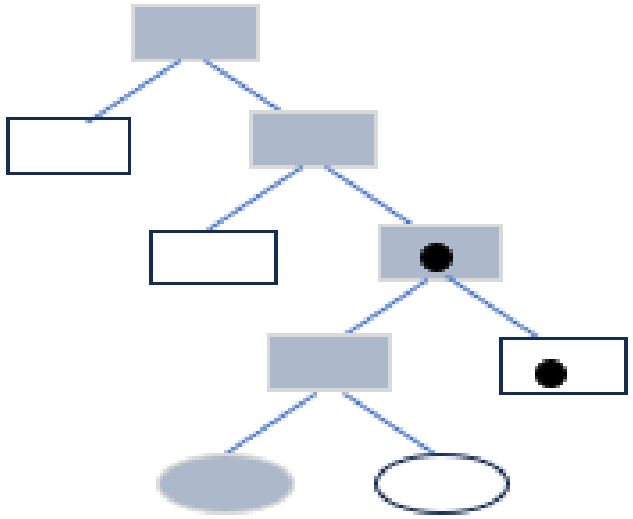


Figure 6.7

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.

XGBoost (Extreme Gradient Boosting)

- Another popular alternative is the XGBoost (Extreme Gradient Boosting). It is in basic terms similar in approach to the GB method, but it offers some extra bells and whistle. XGBoost incorporates various regularization techniques to prevent overfitting and improve the generalization capability of the models. These regularizations occur where optimization procedures are required in the gradient boosting.
- XGBoost employs tree pruning algorithms to control the size of decision trees, reducing overfitting and improving computational efficiency. Tree pruning techniques such as depth-based pruning (to reduce too much depth) and weight-based (to reduce nodes with too little weights) pruning remove unnecessary branches from decision trees. In additional XGBoost employs parallel processing techniques to maximize the computational efficiency.
- XGBoost also provides a built-in support for handling missing values in the dataset during training and prediction. It automatically estimates the best imputation strategy for missing values, and fills in the necessary missing data.

Worked Example

Please upload Chapter6-1.ipynb and follow the computing steps in Jupyter Notebook

- 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 [17].

Gradient Boosting

[17]:
```
from sklearn.ensemble import GradientBoostingClassifier
###   Example:
###   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=1000, 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
```

Worked Example

Please upload Chapter6-1.ipynb and follow the computing steps in Jupyter Notebook

The confusion matrix and the classification report for the test are shown in code lines [18] and [19].

[18]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(y_test, y_pred_GB)
print(confusion_matrix)
```

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

[19]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred_GB))
```

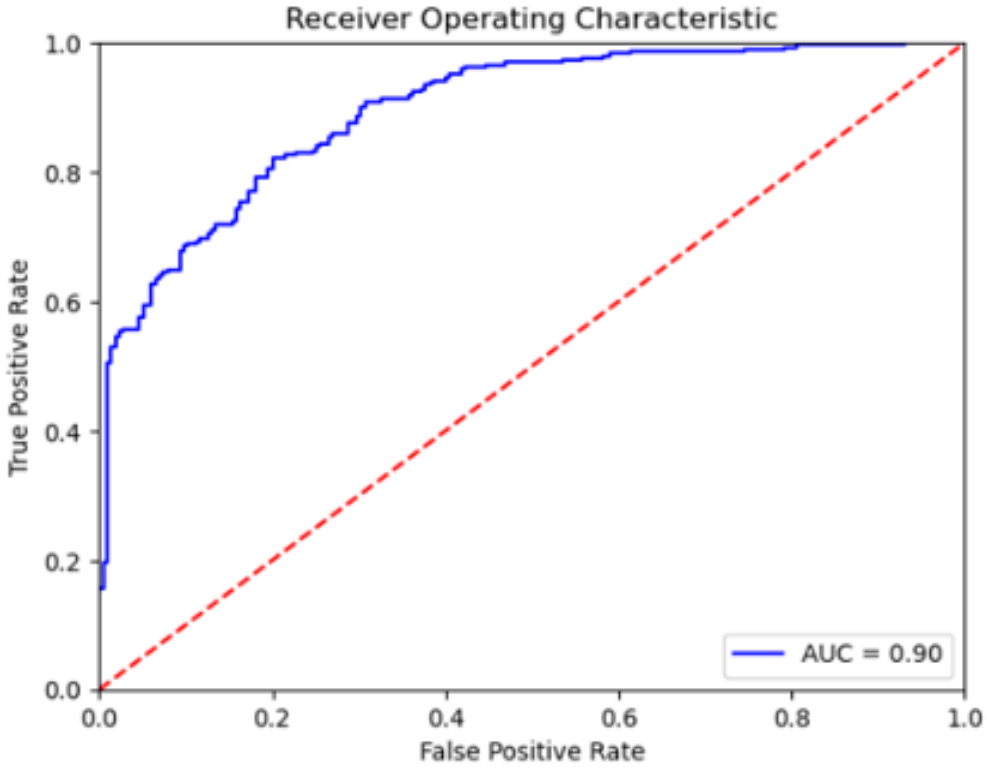
```
              precision    recall  f1-score   support

           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
```

Worked Example

Please upload Chapter6-1.ipynb and follow the computing steps in Jupyter Notebook

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%.



Worked Example

Please upload Chapter6-1.ipynb and follow the computing steps in Jupyter Notebook

We also apply LightGBM. We have to install the lightgbm program from a different source.

The accuracy of lightGBM is 83.27%. The prediction accuracy is higher than GB.

The confusion matrix and classification table are shown in code lines [30], [31].

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

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

```
0.8326771653543307
```

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

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

[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
```

Worked Example

Please upload Chapter6-1.ipynb and follow the computing steps in Jupyter Notebook

We also apply XGBoost. We have to install the xgboost program from another source.

The accuracy of XGBoost is 80.90%. The prediction accuracy is marginally higher than GB.

The confusion matrix and classification table are shown in code lines [37], [38].

[36]:
```
# XGBoost
from xgboost import XGBClassifier
XG_model = XGBClassifier(n_estimators=1000, random_state=1, \
           max_features="sqrt", learning_rate=0.02, max_depth=24, \
           objective='binary:logistic')

### Note for multiclass, objective='multi:softprob'
XG_model.fit(X_train,y_train)

### XG_model follows GB model to have n_estimators = 1000 and max_depth = 24
y_pred_XG = XG_model.predict(X_test)
Accuracy_XG = metrics.accuracy_score(y_test, y_pred_XG)
print("XG Accuracy:",Accuracy_XG)
```

```
XG Accuracy: 0.8090551181102362
```

[37]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(y_test, y_pred_XG)
print(confusion_matrix)
```

```
[[152  63]
 [ 34 259]]
```

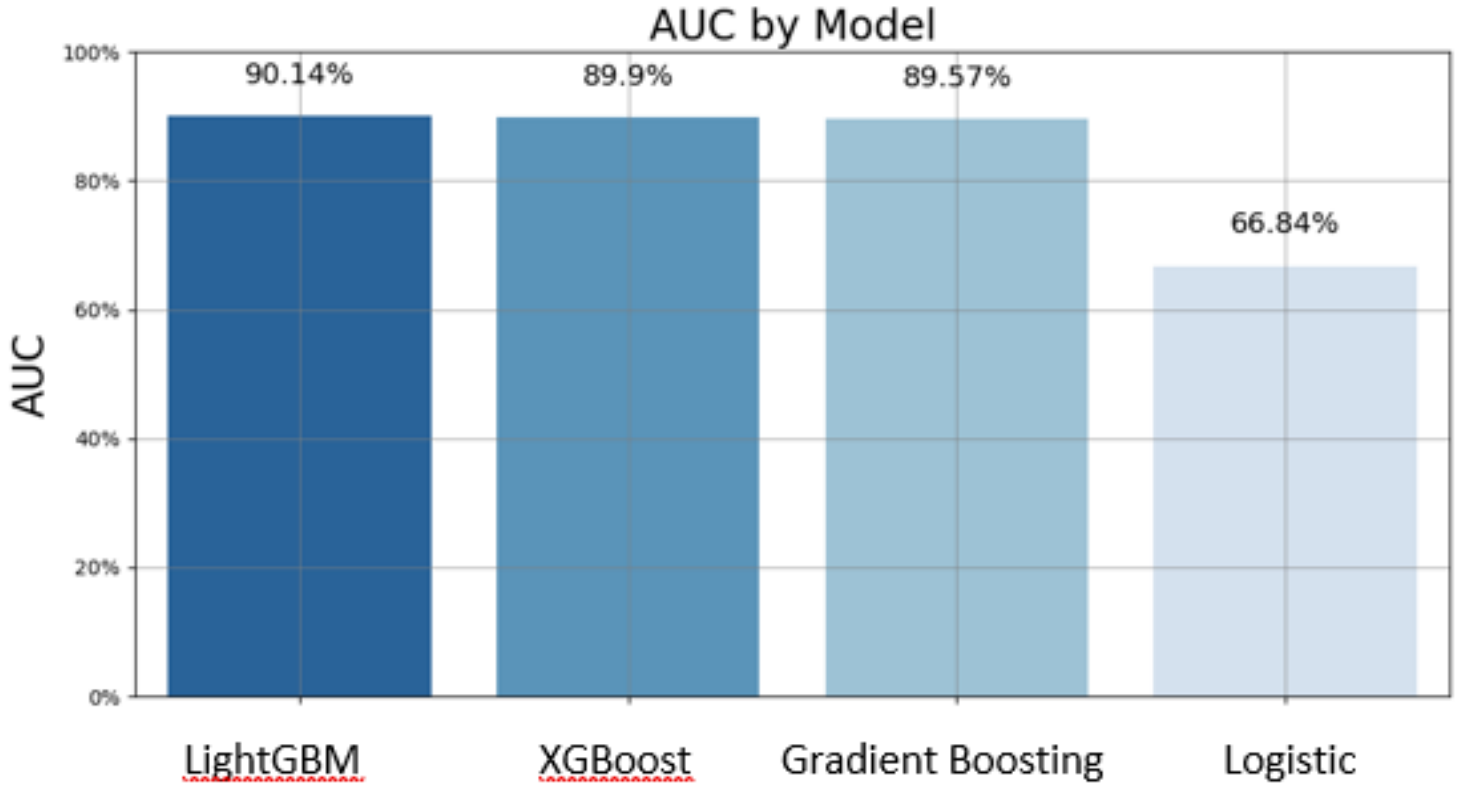
[38]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred_XG))
```

```
              precision    recall  f1-score   support

           0       0.82      0.71      0.76       215
           1       0.80      0.88      0.84       293

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

Worked Example

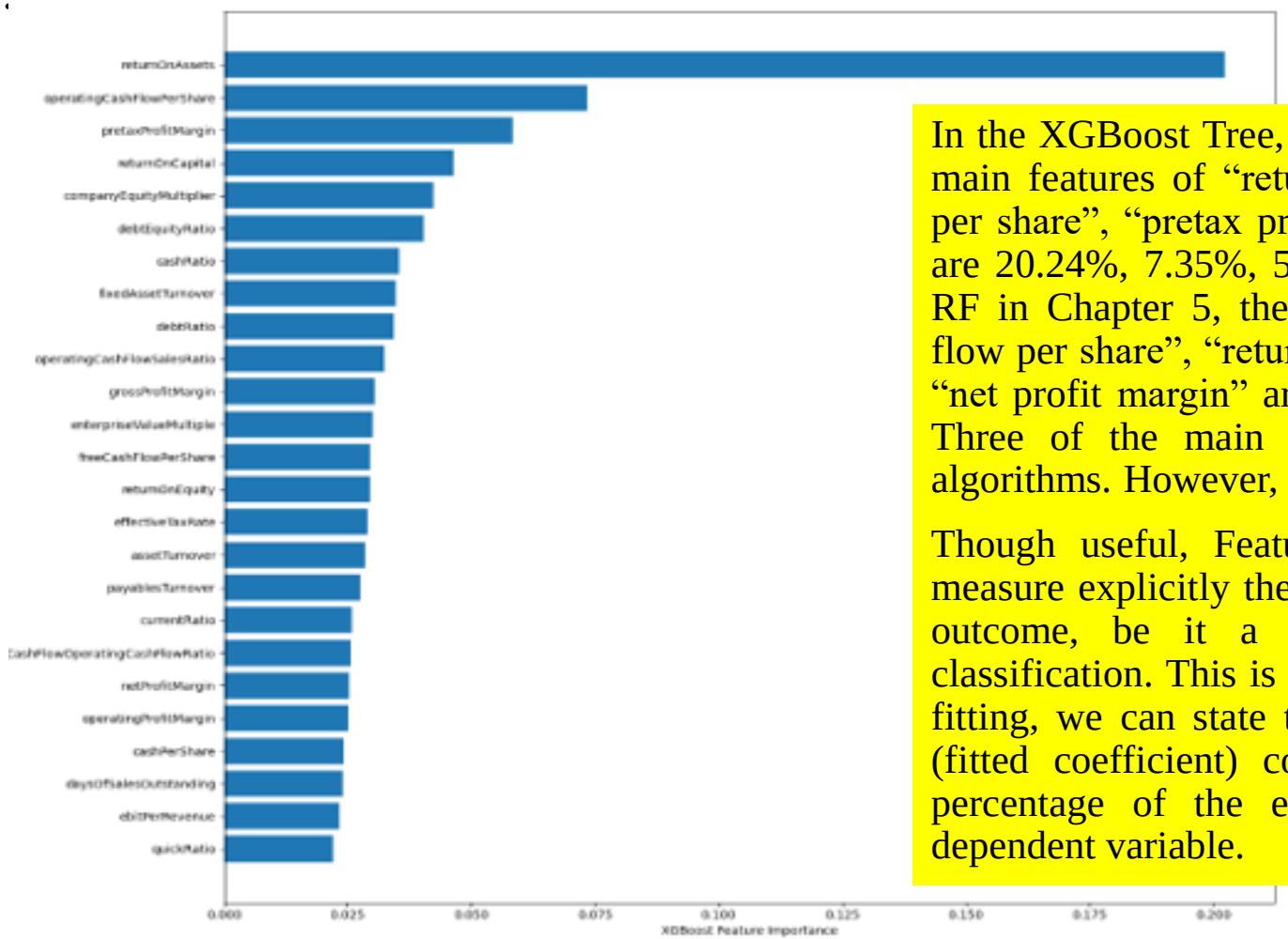
Please upload Chapter6-1.ipynb and follow the computing steps in Jupyter Notebook

The comparison of AUC performances of the various Trees methods and the logistic regression as benchmark are shown as follows.



Shapley and Shap Values

- The percentage contribution of a feature to the overall weighted MSE or Friedman MSE decrease signifies its importance in a GB Tree and its close relatives. In XGBoost, Feature Importance is shown in code lines [51] – [52].



In the XGBoost Tree, the percentage contributions of the 4 main features of "return on assets", "operating cash flow per share", "pretax profit margin", and "return on capital" are 20.24%, 7.35%, 5.83%, and 4.64%. In the case of the RF in Chapter 5, the 4 main features of "operating cash flow per share", "return on assets", "return on capital", and "net profit margin" are 8.06%, 6.06%, 5.75%, and 5.59%. Three of the main features are common to both tree algorithms. However, they are not exactly similar.

Though useful, Feature Importance, however, does not measure explicitly the impact of a feature on the predicted outcome, be it a target value or probability of a classification. This is unlike a linear regression where after fitting, we can state that a feature value times its weight (fitted coefficient) contributes their product value as a percentage of the expected outcome of the target or dependent variable.

Shapley and Shap Values

- We want to consider how much each feature explicitly contributes to the predicted outcome of a ML algorithm. This idea is similar to how much each player in a coalition game contributes to the outcome of a game. This contribution is called the Shapley value. In ML, it is equivalent to the contribution of a feature to the predicted outcome.
- Let each unique player be denoted as player {j} where j is one number of 1, 2, 3,….,N. The problem here is how to determine an efficient and fair allocation $\phi_j(V[N]) \geq 0$ for every player {j}.
- In Shapley game (not other types of games), a coalition [S] consisting of the same set of players will have the same value despite having different ordered sequences of the players. For example, coalition [2] comprising ({7} ∪ {3}) has value V({7} ∪ {3}) = V({3} ∪ {7}). But coalitions [S] with different sets of players can have different values. For example, V({7} ∪ {3}) ≠ V({{8} ∪ {1}) though ({7} ∪ {3}) and ({{8} ∪ {1}) are both coalition [2].
- In the sequence of coalition [1], coalition [2], coalition [3], …… , coalition [N], it is seen that {7}'s marginal contribution is [V({7}) – V([0])] = V({7}) which is observable by the Shapley game. {3}'s marginal contribution is [V({7} ∪ {3}) – V({7})] which is observable by the Shapley game.
- The Shapley Value of player j or {j} refers to his/her fair contribution $\phi_j(V[N])$ to the game payout V([N]) as the average of his/her marginal contributions over all possible sequencings, N!.

$$\phi_j(V[N]) = \frac{1}{N!}\sum_{S=0}^{N-1} S! \times (N-1-S)! \; \{\sum_{k=1}^{C_S^{N-1}} [\mathrm{V}([\mathrm{k,S}] \cup \{\mathrm{j}\}) - \mathrm{V}([\mathrm{k,S}])]\}$$

  where $0 \leq S \leq N-1$.

- Shapley allocation is "fair", satisfying properties of efficiency (no leakage), symmetry, nullity, and linear additivity (or linear scaling in total coalition output).

Shapley and Shap Values

- We want to use the concept in the Shapley game to measure how each feature (player) of a set of N features X (N players) in an algorithm (game) F contributes to the predicted outcome F(X).
- F(.) could be a nonlinear ML algorithm such as RF or NN. Feature space X is made of T number of sample points, $X_{T \times N}$. Each tabular row in X has N feature observations and is associated with a target value. Let one row be (x_1, x_2, x_3, …, x_N). The ML algorithm produces a prediction outcome or output F (x_1, x_2, x_3, …, x_N). We want to find out how much does a feature x_j explain output F (x_1, x_2, x_3, …, x_j, …, x_N).
- Let (z_1, z_2, …., z_S) be a particular permutation choosing S ≤ N-1 elements of (x_1, x_2, x_3, …, x_{j-1}, x_{j+1}, …., x_N) from any row of sample size T. Let (z_{S+2}, z_{S+3}, …., z_N) be a particular permutation choosing N-S-1 elements of the remaining features in (x_1, x_2, x_3, …, x_{j-1}, x_{j+1}, …., x_N) from the same row of sample size T. Then F (z_1, z_2, …., z_S, $\mathbf{x_j}$, z_{S+2}, z_{S+3}, …., z_N) − F (z_1, z_2, …., z_S, $\mathbf{r_j}$, z_{S+2}, z_{S+3}, …., z_N) is equivalent to V({z_1}, {z_2}, …., {z_S}, {x_j}) − V({z_1}, {z_2}, …., {z_S}), where r_j is a randomly selected element from feature j in any of the T rows. The full set of N features is included in computing the marginal contribution of feature $\mathbf{x_j}$ as required in the ML algorithm. In practice, for higher accuracy, the random selection of r_j can be repeated and the average is taken for F (z_1, z_2, …., z_S, $\mathbf{x_j}$, z_{S+2}, z_{S+3}, …., z_N) − F (z_1, z_2, …., z_S, $\mathbf{r_j}$, z_{S+2}, z_{S+3}, …., z_N). To compute Shapley value of feature x_j:

$$\phi_j(V[N]) \equiv \frac{1}{N!}\sum_{S=0}^{N-1} S! \times (N-1-S)! \times \{\sum_{k=1}^{C_S^{N-1}} \left[\mathrm{F}(z_1, z_2, \dots, z_S, x_j, z_{S+2}, \dots, z_N) - \mathrm{F}(z_1, z_2, \dots, z_S, r_j, z_{S+2}, \dots, z_N)\right]\}$$

where 0 ≤ S ≤ N-1. V[N] is the prediction of the target/label for the particular instance or sample point i in the training data set, i.e, V[N] = F(.) for a particular instance i with features in F (x_1, x_2, x_3, …, x_j, …, x_N).

Shapley and Shap Values

- The Shapley value of a feature value is the average change in the prediction when the feature value joins an existing coalition of features. However, the above exact method is computationally too tedious for every x_j as it involves too many C_S^{N-1} (for each $S \leq N-1$) number of combinatorial computations of F(.), especially when the number of features N is large. In practice some approximation methods are used.
- One approximation method is to use Monte Carlo sampling. We want to find the approximate Shapley value/contribution of the j^{th} feature of $x = (x_1, x_2, x_3, \ldots, x_j, \ldots, x_N)$. This is finding feature x_j's Shapley value $\phi_j(V[N])$.
- Let an iteration m involve the following.

  (1) Randomly draw a row from feature matrix $X_{T\times N}$. Call this draw $Z = (z_1, z_2, z_3, \ldots, z_N)$.

  (2) Perform a random permutation of the order of the features – call this permutation "*", but leaving out the j^{th} feature of x_j. For example, $*(z_1, z_2, z_3, \ldots, z_j, \ldots, z_N)$ becomes $(z_5, z_7, z_1, \ldots, z_j, \ldots, z_6)$. This simulates the different sequencings or permutations in Shapley value calculation while keeping x_j intact.

  (3) Form two new instances based on a row of X (going from row 1 to T, or if more data are required, re-sampling some of these rows) and the randomly drawn Z. Permutate these instances based on "*" in step (2). Call these permutated sequences x_{j*}'s and z_{j*}'s.

  (4) Adjust one of the two permutated instances by replacing the j+1, j+2, …, N elements of the permutated X sequence with $z_{j+1*}, \ldots, z_{N*}$. Hence $x_{+j} = (x_{1*}, x_{2*}, x_{3*}, \ldots, x_{j-1*}, x_j, z_{j+1*}, \ldots, z_{N*})$. Adjust the other permutated instance by replacing the j, j+1, j+2, …, N elements of the permutated X sequence with $z_{j*}, z_{j+1*}, \ldots, z_{N*}$. Here, z_{j*} replaces x_j. Hence $x_{-j} = (x_{1*}, x_{2*}, x_{3*}, \ldots, x_{j-1*}, z_{j*}, z_{j+1*}, \ldots, z_{N*})$.

  (5) Compute $\phi_j^m = F(x_{+j}) - F(x_{-j})$. This is marginal contribution of x_j in one constructed iteration.

  (6) Repeat iterations for m = 1, …, M for a large M.
- Compute approximate Shapley value as average: $\phi_j(x) = \frac{1}{M}\sum_{m=1}^{M}\phi_j^m$.

Shapley and Shap Values

- The concept in Shapley method is to measure how each feature (player) of a set of N features X (N players) in an algorithm (game) F contributes to the predicted outcome F(x) when features in x = (x_1, x_2, x_3, …, x_j, …, x_N) are used.
- In Eq. (6.2), suppose the features are arranged as (x_3, x_8, x_5, x_{11} ,…, x_N , x_4 , … , x_7 , x_9) which is some permutation of the original x. This permutation of the N number of represented features will produce contribution of F(x_3, r_8 , r_5 , …, r_N , …, r_9) – F(r_3, r_8 , r_5 , …, r_N , …, r_9) by feature 3, contribution of F(x_3, x_8 , r_5 , …, r_N , …, r_9) – F(x_3, r_8 , r_5 , …, r_N , …, r_9) by feature 8, contribution of F(x_3, x_8 , x_5 , r_{11} , …, r_N , …, r_9) – F(x_3, x_8 , r_5 , r_{11} , …, r_N , …, r_9) by feature 5, ……. , contribution of F(x_3, x_8 , x_5 , …, x_N , r_4 , …, r_9) – F(x_3, x_8 , x_5 , …, r_N , r_4 , …, r_9) by feature N, …….., and contribution of F(x_3, x_8, x_5, x_{11} ,…, x_N , x_4 , … , x_7 , x_9) – F(x_3, x_8, x_5, x_{11} ,…, x_N , x_4 , … , x_7 , r_9) by feature 9 where r_j represents a random draw from feature j.
- Suppose we write the sum of all the above items in a k^{th} permutation as $Z_{(k)}$. Then $Z_{(k)}$ = F (x_3, x_8, x_5, x_{11} ,…, x_N , x_4 , … , x_7 , x_9) – F(r_3, r_8 , r_5 , r_{11} ,…, r_N , r_4 , … , r_7 , r_9). Now suppose F(x) value is independent of the order of the features in x. Then the average across all the N! permutations of x, i.e., $\frac{1}{N!}\sum_{k=1}^{N!} Z_{(k)}$ = F(x) – $\frac{1}{N!}\sum_{k=1}^{N!} F\left(r_{(k)}\right)$ where $r_{(k)}$ is a k^{th} random vector draw from feature vector of X.

Shapley and Shap Values

- With the independence of F(x) from the order of the features in x, $\frac{1}{N!}\sum_{k=1}^{N!} Z_{(k)}$ can also be expressed $\sum_{j=1}^{N} \phi_j(F[x])$ from Eq. (6.2). Then F(x) – $\frac{1}{N!}\sum_{k=1}^{N!} F\left(r_{(k)}\right) = \sum_{j=1}^{N} \phi_j(F[x])$. Let $\phi_0 = \frac{1}{N!}\sum_{k=1}^{N!} F\left(r_{(k)}\right)$.
- Thus, $$F(x) = \phi_0 + \sum_{j=1}^{N} \phi_j(F[x]) \quad (6)$$

  for a particular x = (x_1, x_2, x_3, …, x_j, …, x_N) from a row of X. ϕ_0 can be considered as the expectation of the predictions by F(.) over random x. Eq. (6.3) provides the Shap values $\phi_j(F[x])$ of different j features for the ML algorithm prediction F[x]. The Shap values are additive and together help explain the predicted value of F(x). Shap values also obtain similar nice properties as in the Shapley game.
- This linearized construction still may not help directly in solving for the Shap values ϕ_j. Some ML algorithm such as Trees, may however, provide for each instance that leads up to the final predicted probability the computation of contributions of features in arriving at that predicted probability.

Shapley and Shap Values

- In general, to solve for Shap values, we can implement some approximation methods. For highly nonlinear ML algorithm where finding $\phi_j(F[x])$ directly is a problem due to the enormous amount of computations, approximations of F (x) by a simpler f (x) (with faster computing) in the neighborhood of each x in X may be done. One popular method is called LIME (Local Interpretable Model-agnostic Explanations) model.

- Basically, we search for f (x) that minimizes the loss function L (square error loss) over all sample points i in the neighborhood of each x in X.

$$\min_{f} L(F, f, w) = \Sigma_j^T w_j [F(x_j) - \hat{f}(x_j)]^2 + \Omega(f)$$

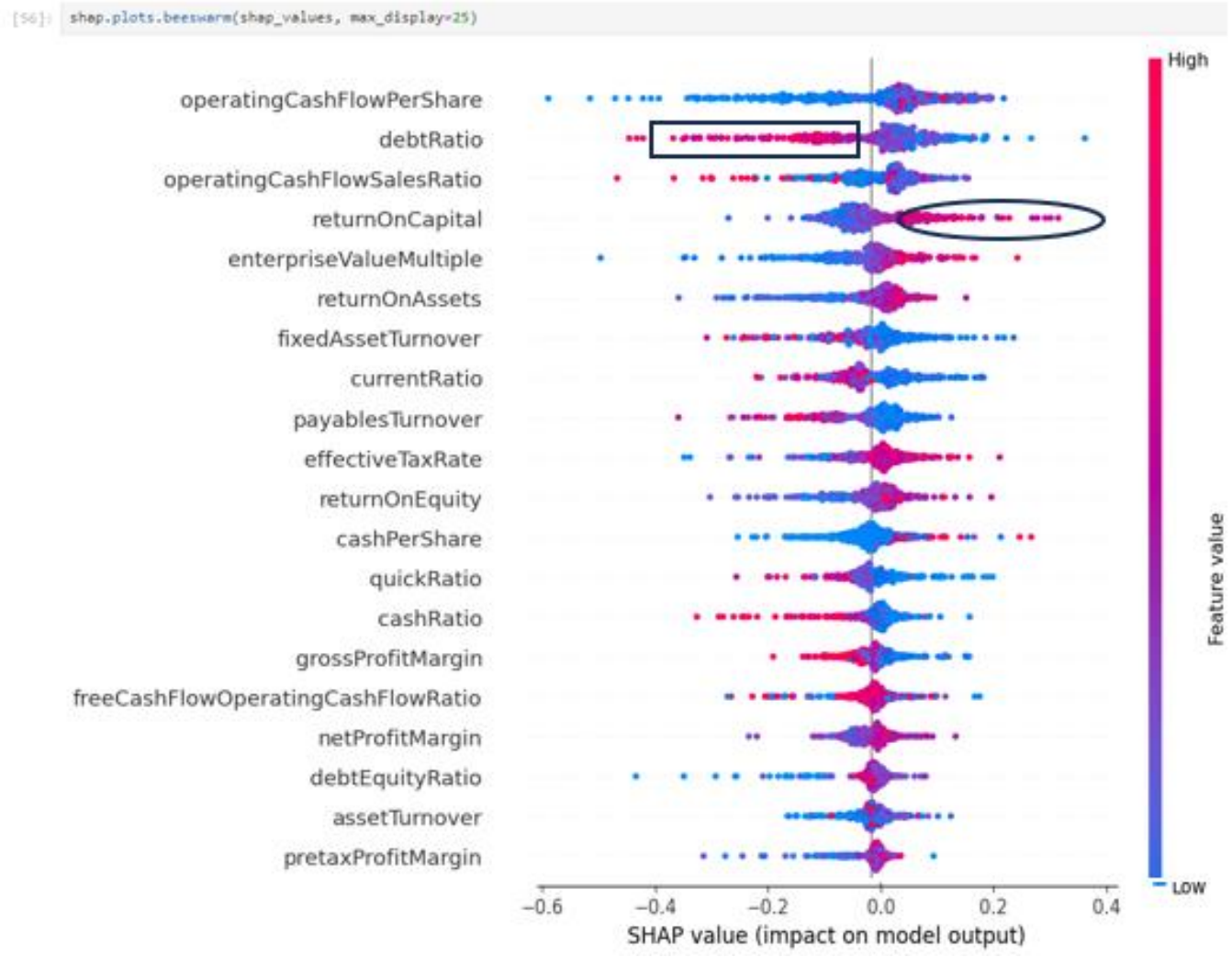
where w is some weighting and Ω(f) is a penalty function that gets larger if the model f is more complex. For example, if f is linear regression, complexity could be reduced by limiting the number of and size of the coefficients. This leads to regularized linear regression. If f is Decision Tree method, complexity could be reduced by limiting the depth of the tree. Typically, the same type of LIME f (x) is used for the X. In principle, as the LIME method can be applied to whatever F(x), it is called a model-agnostic method.

Shapley and Shap Values

- Suppose the prediction or value function is F (x_1, x_2, x_3, …,x_N) for a given row of features x in X. Suppose we approximate F(x) by a linear ridge regression f(x). For each x, randomly generate M local neighborhood points $x^{(1)}$, $x^{(2)}$,….,$x^{(M)}$ where each vector $x^{(i)}$ is Nx1. Weigh more distant point $x^{(i)}$ from original x by smaller weight, e.g., $w^{(i)} = \exp(-||x^{(i)} - x||^2)/D$ where D > 0 is a hyperparameter denoting extent of circle of influence. Larger D means all weights increase. Hence weighted point $x^{(i)}$ becomes $x^{*(i)} = w^{(i)}x^{(i)}$.
- Using the ML algorithm F(.), find $F(x^{*(1)}) = Y_1$, $F(x^{*(2)}) = Y_2$, ……, $F(x^{*(M)}) = Y_M$. Then perform linear regression $Y_i = b_0 + b_1 x_1^{*(i)} + b_2 x_2^{*(i)} + ….. + b_N x_N^{*(i)} + e_i$ for i = 1,2,…,M, with coefficient constraints.
- The regression produces estimates $\hat{b}_0$, $\hat{b}_j$'s from $x^{(i)}$'s. Then find $f(x) = \hat{b}_0 + \sum_{j=1}^{N} \hat{b}_j x_j$ with the original x_j's. For this instance or sample point x, we can take expectation over the probability space of X in that neighborhood to obtain $E[f(X)] = \hat{b}_0 + \sum_{j=1}^{N} \hat{b}_j E(x_j)$.
- Then, $f(x) - E[f(X)] = \sum_{j=1}^{N} \hat{b}_j [x_j - E(x_j)]$ or $f(x) = E[f(X)] + \sum_{j=1}^{N} \hat{b}_j [x_j - E(x_j)]$.
- We can treat $E[f(X)] = \phi_0$ and $\sum_{j=1}^{N} \hat{b}_j [x_j - E(x_j)] = \sum_{j=1}^{N} \phi_j$ as in Eq. (6.3), thus obtaining $\hat{b}_j [x_j - E(x_j)]$ as the j^{th} feature Shap value for instance x. The sum of the Shap values from all the N features in this instance x gives $\sum_{j=1}^{N} \phi_j = f(x) - E[f(X)]$.

Shapley and Shap Values

- Continuing with Chapter6-1.ipynb, we analyse the XGBoost predictions on the test data set of 508 sample points.



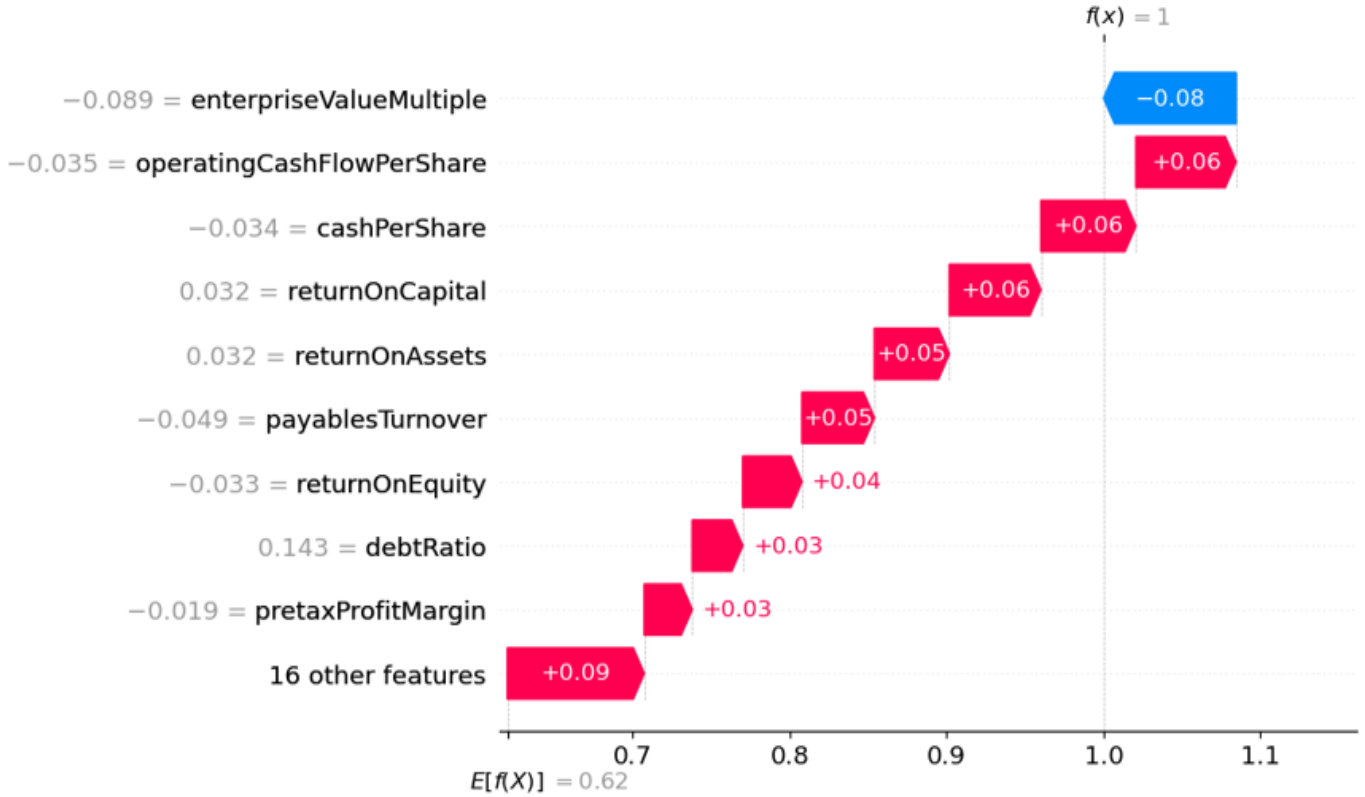
Shapley and Shap Values

Code line [57] shows the tabulation of average absolute Shap value per instance across all the features. It is seen that “operating cash flow per share” and “debt ratio” have the largest contributions to predicted probability of being Class “1”. “operating cash flow per share” is also one of the 4 dominant features in features importance. Note that only the partial graph is shown due to space constraint.

```
[57]: shap.plots.bar(shap_values,max_display=25)
      ### SHAP bar plot will take the mean absolute value of each feature
      ###  over all the instances (rows) of the dataset.
```

Shapley and Shap Values

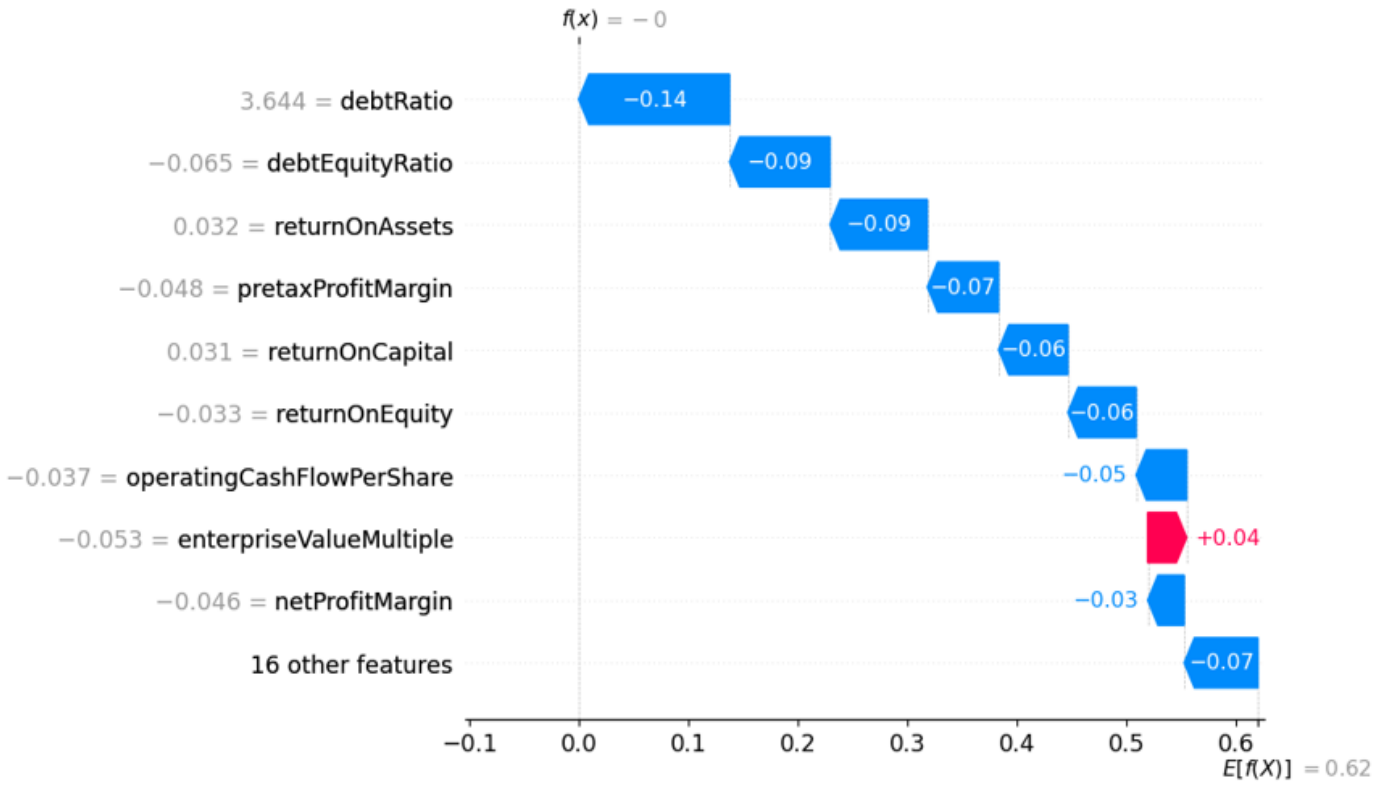
```
[59]: ### waterfall plot for instance or firm [i] or shap_values[i]
      shap.plots.waterfall(shap_values[0])
```



We also show in code line [59] the 'waterfall' plot of the Shap values of features in contribution to the predicted probability of Class "1" of the first firm or instance in the test data set. In this case, the predicted probability f(x) = 1. The sum of Shap values of all the 25 features is equal to f(x) – E[f(x)] = 1 – 0.62 = 0.38 where 0.62 is the % or mean of the predicted values of "1"s in Class "1". For this first firm, the main features with highest Shap value contribution to the predicted probability of Class "1" are "operating cash flow per share", "return on capital", "return on assets", etc. The negative contribution of "enterprise value multiple" is due to a lower than average multiple.

Shapley and Shap Values

```
[65]: ### waterfall plot for instance or firm [i] or shap_values[i]
      shap.plots.waterfall(shap_values[507])
```



We show in code line [65] the 'waterfall' plot of the Shap values of features in contribution to the predicted probability of Class "1" of the last firm or instance in the test data set. In this case, the predicted probability f(x) = 0. The sum of Shap values of all the 25 features is equal to f(x) – E[f(x)] = – 0.62. For this last firm, the main features with largest negative Shap value contribution to the predicted probability of Class "1" are "debt ratio", "debt-equity ratio", "return on assets" (low feature value), etc.

Summary

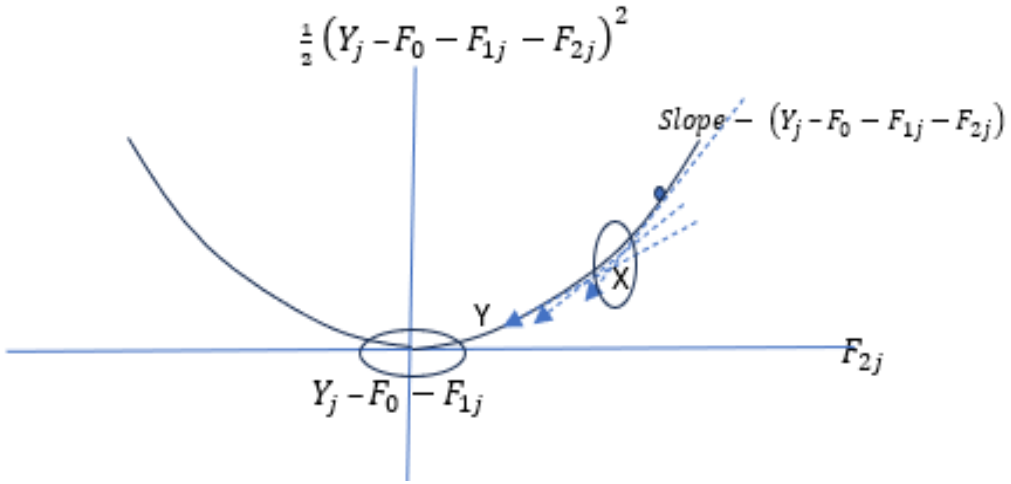
- Ensemble methods include the bagging approach of RF and the gradient boosting approach of GB, LightGBM, and others. Each DT in a RF of say 500 trees produces a prediction x_i. The prediction of the trees used the same decision rules formulated in the training sample for each of the ensemble of trees. 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 $var\left(\frac{1}{500}\sum_{i=1}^{500} x_i\right) \approx \frac{1}{500}\, var\left(x_i\right)$ 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.

Summary

- 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.

Summary

- 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.
- we also see how the Shap values can contribute a better understanding or explanation of the prediction in every instance, in addition to feature importance computations in trees. Specific additions or subtractions to predicted probability by features can help decision-making in improving features to directly affect the probability.

In Class Practice Exercise (not graded):

Chapter6-2.ipynb

Use the data set – Financial_ratios_Industry_new2.csv. This is essentially the same data used in the prediction of industry sector average net profit margin increase (or decrease) based on 70 features and 2763 cases of month-industry average accounting variables.

In this exercise, use all the 70 features in the Random Forest, Gradient Boosting, Light GBM and also the Logistical Regression (the convergence somehow is not affected with the latter, unlike in statsmodel – sklearn may avoid putting up red flags so easily, although there is still the multi-collinearity issue.)

Use 75% training and 25% testing. For RF, use n_estimators = 500, max_features = 'sqrt', random_state=1, max_depth=34. For GB, n_estimators = 500, max_features = 'sqrt', random_state=1, learning_rate=0.055, max_depth=24. For Logit, use random_state=1.

Find their (3 methods): prediction accuracy, confusion matrix (2 x 2 table), and AUC (area under ROC curve).

End of Class