Tree Based Methods in Machine Learning

Tuğrul Hasan Karabulut 1

Contents

1 Introduction

Tree based methods are used across many Machine Learning tasks. They are favored because of their interpretability and their ability in capturing non-linear relationships in the data. Decision tree is simplest among all tree based models. It's very interpretable and straightforward. But it has its disadvantages. Because of its non-parametric nature, it heavily relies on data. Different data may result in completely different trees in the tree building process. This problem is referred as 'a model having high variance'. For this reason, decision trees are non-stable models. They usually fail to generalize, therefore perform poorly on unseen data. In another words, they overfit.

Ensemble methods overcome this issue by combining multiple trees (learners, generally) into a robust model that generalizes well and have high performance on unseen data. They achieve this by reducing the bias (it can be seen as a model's 'unability' of capturing the complexity of the data) or reducing the variance of a model.

In this article, we'll talk about decision trees and ensemble methods that uses decision trees in the context of classification.

2 Decision Trees

2.1 What is a Decision Tree?

Decision trees can seen as a set of if-then rules. Starting from a root node, at each node, a decision is made. Data is splitted to different branches at each decision. At the bottom of the tree, there are leaves. Each member of the data eventually reaches a leaf. At each leaf, a final prediction is made. For example, if we're trying to predict house prices, prediction at a leaf may be the mean (or median) of all house prices in that leaf. If we are making a classification, such as classifying some pictures as cats and dogs, then prediction at a leaf is taking the most common class in that leaf.

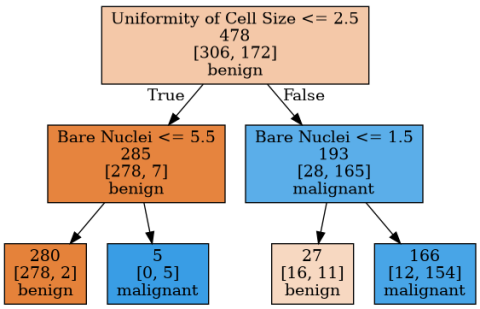

Figure 1: A Simple Decision Tree

In Figure 1, there is a decision tree built based on Wisconsin Breast Cancer dataset from UCI Machine Learning Repository. At each node, there is a condition that splits the data, for example 'Uniformity of Cell Size ≤ 2.5' or 'Bare Nuclei ≤ 1.5'. Nodes at the bottom are leaves that's classifying the tumors that reaches them as benign or malignant.

We see that, at the left most leaf, there are $278+2 = 280$ members (tumor records). These are from training data that the tree is built from. The model classified this leaf as benign because the majority class was benign. Every new observation that falls into that leaf will be classified as benign. 280 members of the training data reached this leaf. 278 of them were benign and 2 of them were malignant. So, it's logical to label this leaf as benign because it's the majority class. On the other hand, second leaf from the right does not look good. There are 16 benign and 11 malignant tumors that fell into that leaf and because we label the leaf as whatever the majority class is, it's been labeled as benign. But the frequencies of benign and malignant are very close. The node is non-homogeneous. Maybe it needed further splits.

Another thing that we might have been asked is, in the root node, for example, how

did we decide to split the node by the condition 'Uniformity of Cell Size ≤ 2.5'? How did we determine that 2.5 boundary? There must be some mechanism that help us decide the 'best split' in a specific node.

So, there should be two important aspects to consider in tree building process. At each node, we check if it might be a good idea to split that node further and find what is the best possible split criterion that creates homogeneous (pure) child nodes.

How can we measure the impurity of a node? One possible measurement is *cross-entropy*, which is defined as:

$$\sum_{i=1}^{K} -p_i \log p_i \tag{1}$$

where K is the number of classes, 2 for the tumor case. p_i is the proportion of class i in the node and defined as:

$$p_i = \frac{N_i}{N}$$

where N_i is the number of members that belong to class i in the node and N is total members in the node.

Another impurity function is *gini coefficient*:

$$1 - \sum_{i=1}^{K} p_i^2 \tag{2}$$

So, we calculate the impurity of a node by an impurity measurement function. We want to create pure leaf nodes. To do that, we must select at each splitting stage, the split that reduces the impurity most. We need to calculate the impurity after the split.

Suppose that a splitting criterion, C_i, splits a node into n nodes. Choosing the cross-entropy as our impurity measure, impurity after the split is defined as:

$$\mathrm{I}' = -\sum_{j=1}^{n} \frac{N_j}{N} \sum_{i=1}^{K} p_{ij} \log p_{ij} \tag{3}$$

We choose the split, C, that reduces the cross-entropy most. In other words, at a node m, we're looking for a C that maximizes the cross-entropy decrease, which is defined as:

$$D(m) = \mathrm{I_m} - \mathrm{I_m^{'}} \tag{4}$$

where $\mathrm{I_m}$ is from (1) and $\mathrm{I'_m}$ is from (3). They are calculated based on the node m.

With these fundamental ideas in our mind, let us further explore decision trees by looking at the tree building process.

2.2 How to Build a Decision Tree

There are different algorithms for decision tree building. But their core ideas are same. They only differ in data type treatment, tree structure and some additional heuristics.

Suppose that we have n explanatory variables. If an explanatory variable x_i is categorical and has M distinct values, there are $2^{M-1}-1$ different splits. An example split would be $C(m) = 1(x_i = m)$ where m is a possible value that x_i can have. $1(.)$ is the indicator function that returns 0 or 1 based on the boolean expression that it receives as argument. So, the example split, splits the node into two different sets. Therefore, it creates a two different node. If x_i is a numerical (continuous) variable, it can have infinite number of values. But because we can not search for every possible numerical value, we look for the boundary values in our training data. So, that leaves us out with $M - 1$ possible splits. For example, suppose that we have a training set that contains N records and suppose that we have an explanatory variable x_i and has values $S_i = \{x_i^{(j)}\}_{j=1}^{N}$ such that S_i is sorted. We check the possible splits, $C_i(m) = 1(x_i < m)$ where $m = \dfrac{x_i^{(j)} + x_i^{(j+1)}}{2}, j = 0, 1, 2, ..., m - 1.$ This form of split, again, splits a set into two distinct sets. These are the basic splitting criteria that we will use.

Let us start with the simplest one, the CART algorithm. CART uses the gini index as splitting criterion. Also, CART treats every variable as numerical. So, it always look for split in the form $C(m) = 1(x < m)$. Hence, it always does binary splitting. So, it creates binary trees. Starting from a single node, for each variable, it finds the best split $C_i(m)$, and selects the best split among these variables. In another words, it selects the split that maximizes the impurity decrease from (2). And it recursively repeats this process until there is no decrease in the impurity or the impurity of the

current node is less than some threshold value.

Algorithm 1: Decision Tree Building with CART Algorithm for Classification

Input: A training set X
Output: Decision Tree
Function `BuildTree(`X`):`
    /* equation (2) */
    while $Impurity(X) > threshold$ **do**
        $BestFeature,\ BestSplit \leftarrow$ `FindBestSplit(`X`)`
        Split X into X_{left}, X_{right} using $BestFeature$ and $BestSplit$
        `BuildTree(`X_{left}`)`
        `BuildTree(`X_{right}`)`
    end
    return;
Function `FindBestSplit(`X`):`
    $MaxImpurityDecrease \leftarrow 0$
    /* n is the number of explanatory variables */
    for $i \leftarrow 1$ ***to*** n **do**
        foreach *possible split* m *for* x_i **do**
            Split X into X_{left}, X_{right} using m, where
            $X_{left} = \{x \in X \mid x_i < m\}, X_{right} = \{x \in X \mid x_i \geq m\}$
            /* equation (4) */
            $ImpurityDecreaseAfterSplit \leftarrow ImpurityDecrease(m)$
            if $MaxImpurityDecrease < ImpurityDecreaseAfterSplit$ **then**
                $MaxImpurityDecrease \leftarrow ImpurityDecreaseAfterSplit$
                $BestFeature \leftarrow i$
                $BestSplit \leftarrow m$
            end
        end
    end
    return $BestFeature,\ BestSplit$

Basic CART algorithm is given above. Only one stopping criterion is given in the algorithm. It stops when the current node's impurity is less than or equal to some user-defined threshold. But there are other stopping criteria as well. For example, enforcing a maximum depth to the tree. When the tree reaches to a specified maximum depth, the algorithm stops. Or specifying the minimum members at a node to make it a leaf. When a node has less than or equal number of members inside it, the algorithm stops. These stopping criteria prevents the tree from growing too large. Consequently, prevents overfitting. These criteria often referred as *pre-pruning*, meaning that pruning the tree while building it. There are also *post-pruning* techniques which prunes the free after it is grown. We won't discuss post-pruning in this article.

2.3 Other Algorithms for Tree Building

There are other decision tree building algorithms besides CART. Some examples are *ID3*, *C4.5*, *C5.0*, etc,. For example, C4.5 treats categorical and numerical variables differently. It does binary splitting for numerical variables by finding a threshold as in CART. It does K-way splitting for categorical variables that can have K distinct values. Also, it uses cross-entropy as splitting criterion.

Now, let us discuss how to fully exploit decision trees by applying ensemble methods.

3 Ensemble Learning

Basic idea behind the ensemble learning is combining multiple base learners to create a powerful model that has higher performance than each individual learner's performance. There are two popular methods for ensemble learning: *bagging* and *boosting*. Bagging works by training multiple learners on a sample drawn from the training set. And in prediction time, it takes the average of each learner's prediction (for regression) or it take the majority vote (for classification). Boosting has a different approach. It starts with a 'weak learner' that performs slightly better than random guessing. And it iteratively adds new weak learners to the model to fix the errors that the previous learners made.

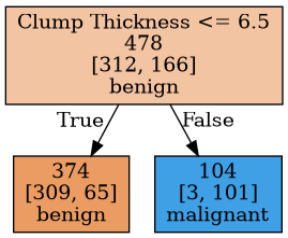

Figure 2: A Decision Stump

Decision trees are often used as base learners in ensemble methods. In bagging, multiple large trees with high variance are trained on different samples of the training set in order to create a robust model that has low variance. In boosting, decision stumps or small trees are often used as high biased learners. A decision stump is a decision tree which goes only one level deep. It does only one decision. In Figure 1, there is an example decision stump built from wisconsin breast cancer data set. It only uses the feature 'Clump Thickness' and makes a single split.

3.1 Bagging

The term bagging stands for '*bootstrap aggregation*'. Let's define *bootstrapping* and *aggregating*. Bootstrapping is any method that uses *random sampling with replacement*, which means some sample may have repeated observations. Aggregating means combining the results taken from the different samples.. In the regression case, 'combining' means is taking the average of the results (5). In classification, it means taking the majority vote (6). It turns out that taking bunch of samples with replacement, training some models on them and aggregating the results has a variance reducing effect. Therefore, by bagging, a model with low variance model can be obtained by using high variance models.

$$F_R(x) = \frac{1}{m} \sum_{i=1}^{m} f_i(x) \tag{5}$$

$$F_C(x) = \underset{j}{\operatorname{argmax}} \sum_{i=1}^{m} 1(f_i(x) = j), \ j = 0, 1, \ldots, K \tag{6}$$

m is the number of trained models.

3.2 Random Forest

Random Forest is an ensemble learning algorithm that leverages bagging and decision trees. Decision trees are great choice for ensemble methods because they usually have high variance. Multiple decision trees can be used together to both reduce their individual variances and make use of their power in capturing non-linear relationships in the data.

Besides using decision trees, Random Forest does more one thing to create less correlated tree to reach more predictive performance. If some features in our data set are more correlated with our target features, then every decision trees will use those features to make prediction and therefore, all the trees will be correlated with one another. So, we would end up with trees that are mostly identical to another and our overall model would have low performance. What Random Forest does to prevent this problem is this: For each tree, it uses only a portion of the features in the data set rather using all features. Number of trees and number of features to

uses at each tree building process are hyperparameters to tune.

Algorithm 2: Random Forest Algorithm For Classification

Input: A training set X, number of trees to grow M, number of features to use for each tree n'

Output: Aggregator function F_C

for $i \leftarrow 1$ ***to*** M **do**

  Draw a sample X' from the training set X with replacement

  Select n' features from all n features at random.

  Train a decision tree using X' and n' and call it $f_i(x)$

end

return $F_C(x) = \underset{j}{\operatorname{argmax}} \sum_{i=1}^{m} 1(f_i(x) = j),\ j = 0, 1, \ldots, K$

3.3 Boosting

Boosting is an ensemble method that takes a weak learning algorithm and builds a strong predictor in a forward stagewise fashion. It starts with an initial guess f_0, and iteratively adds new weak learners with the objective of reducing the error of the current model. There are several techniques for reducing the error. Some examples of error reducing techniques are reweighting or resampling the training set so that the new learner would be forced to focus on the examples with large errors (hard examples). Other unique technique is Gradient Boosting, which makes use of numerical optimization in the function space of weak learners.

Boosting creates additive models. And it does that in an iterative way. At each stage, a weak learner is built according to the current overall model's errors. An additive model has the following form:

$$F(x) = f_0 + f_1(x) + \cdots + f_m(x) = f_0 + \sum_{i=1}^{m} f_i(x) \tag{7}$$

Every f_i is the resulting function of a weak learner. Weak learner might be a parametric regression model with small amount of parameters or a small decision tree, etc.

$$f_i(\mathbf{x}; \theta_\mathbf{i}) = \theta_{i0} + \theta_{i1}x_1 + \cdots + \theta_{in}x_n \tag{8}$$

$$f_i(\mathbf{x}; \mathbf{w_i}, \{R_j\}_{j=1}^{J}) = \sum_{j=1}^{J} w_{ij} 1(\mathbf{x} \in R_j) \tag{9}$$

In equations (8) and (9), functions learned from linear regression and decision trees are given, respectively. In (8), there is a linear regression model with n features. In (9), a decision tree model which has J terminal nodes (leaves) is given. Each region that corresponds to a leaf is given as R_j and w_{ij} is the prediction at the jth leaf.

Each boosting technique is actually doing a forward stagewise additive modelling which is iteratively improving our overall model with small models by choosing a model which reduces our loss, L. Its general algorithm is given below.

Algorithm 3: Forward Stagewise Additive Modelling

Output: An additive model F_M

$F_0(x) = f_0$

for $m \leftarrow 1$ ***to*** M **do**

$$(\beta_m, f_m) = \underset{\beta,\gamma}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, F_{m-1}(x) + \beta f(x;\gamma))$$

$$F_m(x) = F_{m-1}(x) + \beta_m f_m(x;\gamma_m)$$

end

return $F_M(x)$

Now, let's talk about the AdaBoost algorithm.

3.4 AdaBoost

AdaBoost is the first popular boosting algorithm. It uses multiple weak learners which each weak learner focuses on the errors that the previous weak learner has made. It does that by assigning weights to the observations based on some error criterion. Resampling is also used instead of weighting, which does random sampling but gives higher probabilities to the hard examples in order to select the observations which has greater error. We'll talk about the reweighting case.

It starts with a weak learner and weights $w_i = \frac{1}{N}$ where N is the number of observations in our training set. After each stage, weights are modified based on the errors of individual observations. Observations with high errors have high weights whereas weights of the observations that are correctly predicted are decreased. Next weak learner is trained using those weights. Therefore, at each stage, observations that are hard to predict correctly gets special treatment.

Now, let us formulate the AdaBoost for binary classification.

Let F_{m-1} denote the sum of the weak learned that are fitted in the previous stages.

$$F_{m-1}(\mathbf{x}) = f_0 + f_1(\mathbf{x}) + f_2(\mathbf{x}) + \cdots + f_{m-2}(\mathbf{x}) + f_{m-1}(\mathbf{x}) \tag{10}$$

where each f_i is a decision tree of the form given in (9).

Furthermore, let us define a loss function $L(y, f(x))$ where y is the ground truth and $f(x)$ is the prediction obtained through a boosted model as in (10). Also, suppose that $y \in \{-1, 1\}$

AdaBoost uses exponential loss criterion which is defined as:

$$L(y, f(x)) = \exp\left(-yf(x)\right) \tag{11}$$

At each stage, AdaBoost must solve:

$$(\beta_m, f_m) = \underset{\beta,f}{\operatorname{argmin}} \sum_{i=1}^{N} \exp(-y_i(F_{m-1}(x) + \beta f(x_i)))$$

where f_m is the weak learner that is to be learned and $\beta_m > 0$ is its coefficient which controls its influence in the overall model.

We can simplify the objective above as the following:

$$(\beta_m, f_m) = \underset{\beta,f}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} \exp(-y_i \beta f(x_i)) \tag{12}$$

where $w_i^{(m)} = \exp(-y_i F_{m-1}(x))$. w_i^m's are not related to β and f, so we can see them as weights. Solution of this objective involves two steps. First, for any β, we have:

$$f_m(x) = \underset{f}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} 1(\ y_i \neq f(x_i)\ )$$

Now let us find β_m. We can further simplify the objective given in (11) by separating the summation into two summations based on $y_i = f(x_i)$ and $y_i \neq f(x_i)$. If $y = f(x)$, then $exp(-yf(x)) = e^{-1}$, otherwise it is e^1. Therefore it can be written as:

$$e^{-\beta}. \sum_{y_i = f(x_i)} w_i^{(m)} + \ e^{\beta}. \sum_{y_i \neq f(x_i)} w_i^{(m)}$$

To find the β_m that will minimizes this equation, we take derivative with respect to β and set it to zero:

$$-e^{-\beta}. \sum_{y_i = f(x_i)} w_i^{(m)} + \ e^{\beta}. \sum_{y_i \neq f(x_i)} w_i^{(m)} = 0$$

When we pull β from the above equation, we find:

$$\beta_m = \frac{1}{2} \log \frac{1 - err_m}{err_m}$$

where

$$err_m = \frac{\sum_{i=1}^{N} w_i^{(m)} 1(y_i \neq f_m(x_i))}{\sum_{i=1}^{N} w_i^{(m)}}$$

Finally, we update our prediction as:

$$F_m(x) = F_{m-1}(x) + \beta_m f_m(x)$$

Also, recall that our weights were:

$$w_i^{(m)} = exp(-y_i F_{m-1}(x))$$

In the next iteration, new weights will be:

$$\begin{aligned} w_i^{(m+1)} &= exp(-y_i F_m(x_i)) \\ &= exp(-y_i(F_{m-1}(x_i) + \beta_m f_m(x_i))) \\ &= exp(-y_i F_{m-1}(x_i)) \ . \ exp(-y_i \beta_m f_m(x_i)) \\ w_i^{(m+1)} &= w_i^{(m)} \ . \ e^{-y_i \beta_m f_m(x_i)} \end{aligned}$$

After updating our weights, we normalize them so that their sum equals to 1:

$$w_i^{(m+1)} := \frac{w_i^{(m+1)}}{\sum_{k=1}^{N} w_k^{(m+1)}}$$

Note that we can write $-yf(x)$ as $2(\ 1(y = f(x))\ ) - 1$, therefore we can change our weight update rule by the following:

$$\begin{aligned} w_i^{(m+1)} &= w_i^{(m)} \ . \ e^{\beta_m [2(1(y_i = f_m(x_i))) - 1]} \\ &= w_i^{(m)} \ . \ e^{\alpha_m 1(y_i = f_m(x_i)) - e^{\beta_m}} \end{aligned}$$

where $\alpha_m = 2\beta_m$. Also, β_m is common for all $w_i^{(m)}$. So, we can simplify our update rule as following:

$$w_i^{(m+1)} = w_i^{(m)} \; . \; e^{\alpha_m 1(y_i = f_m(x_i))} \tag{13}$$

Algorithm is given below.

Algorithm 4: AdaBoost For Binary Classification

Output: An AdaBoost model $F(x)$

$F(x) = 0$

Initialize weights $w_i^{(0)} \leftarrow \frac{1}{N}$

for $m \leftarrow 1$ ***to*** M **do**

Fit a classifier f_m using weights $w_i^{(m)}$

Compute $err_m \leftarrow \frac{\sum_{i=1}^{N} w_i^{(m)} 1(y_i \neq f_m(x_i))}{\sum_{i=1}^{N} w_i^{(m)}}$

Compute $\alpha_m \leftarrow \log \frac{1 - err_m}{err_m}$

Update weights $w_i^{(m+1)} \leftarrow w_i^{(m)} \; . \; e^{\alpha_m 1(y_i = f_m(x_i))}$

Normalize weights $w_i^{(m+1)} \leftarrow \frac{w_i^{(m+1)}}{\sum_{k=1}^{N} w_k^{(m+1)}}$

end

return $F(x) = sign\left[\sum_{m=1}^{M} \alpha_m f_m(x)\right]$

Final model is just the weighted majority vote of the trained classifiers. In the same logic, the predictor multi-class can be defined as:

$$F(x) = \underset{k}{\text{argmax}} \left(\sum_{i=1}^{m} \alpha_m \; . \; 1(f_m(x) = k) \right)$$

where $k = 0, 1, \ldots, K$

3.4.1 Why Exponential Loss?

AdaBoost uses exponential loss criterion given in (11). One advantage of exponential loss is its low computational cost. This makes it an appropriate choice for additive models like AdaBoost becasue of their iterative training process.

Let us now see why exponential loss function works and how to minimize it.

We, again, will analyze the binary classification case where $y \in \{-1, 1\}$. Suppose that we have a predictor $f(x)$.

Exponential loss is defined as:

$$L(y, f) = e^{-yf(x)}$$

We want to minimize the expected loss $E_{Y|x}[e^{-yf(x)}]$ where Y is a discrete random variable that takes values in { -1, 1 }. Expected loss is defined as:

$$\begin{aligned} E_{Y|x}[e^{-yf(x)}] &= \sum_y e^{-yf(x)} P(Y = y|x) \\ &= P(Y = +1|x)e^{-f(x)} + P(Y = -1|x)e^{f(x)} \end{aligned}$$

We want to find the predictor $f^*(x)$ that minimizes this loss function. So, we take derivative with respect to f and set it equal to 0.

$$-P(Y = +1|x)e^{-f(x)} + P(Y = -1|x)e^{f(x)} = 0$$

After pulling f from the equation, we find f^* as:

$$f^*(x) = \frac{1}{2} \log \frac{P(Y = +1|x)}{P(Y = -1|x)}$$

which is one half of the *log-odds* (or *logit*) function. This result allows us to make sense of exponential loss because when $P(Y = +1|x) > 0.5$, logit function gives a positive value, and it gives a negative value if $P(Y = +1|x) < 0.5$ (or $P(Y = -1|x) > 0.5$) (See Figure 3). Recall that our prediction function was the *sign* function (Algorithm 4) in binary classification case of AdaBoost. So, this convince us of the choice of exponential loss function becasue whenever we have a probability less than 0.5, it returns the negative class, otherwise it returns the positive class.

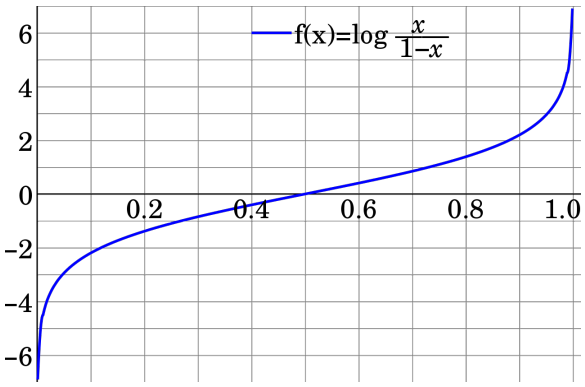

Figure 3: Logit Function

3.5 Gradient Boosting

In a general supervised machine learning setting, we often want to find an estimate $F(\mathbf{x})$, that produces a value y^* as the prediction to the ground truth y. To find the optimal F, we find the function that minimizes the expected value of some pre-determined loss function.

$$F^* = \underset{F}{\operatorname{argmin}}\ E_{Y|x}[L(y, F(\mathbf{x}))] \tag{14}$$

Instead of looking at all possible functions, we usually narrow our function space down to a family of parameterized functions $F(\mathbf{x}; \mathbf{P})$ where $\mathbf{P}$ is a set of parameters:

$$\mathbf{P} = \begin{bmatrix} P_1 \\ P_2 \\ \vdots \end{bmatrix}$$

Now, the problems reduces to finding the parameters, P^*, that minimizes the expected loss.

$$\begin{aligned} \mathbf{P}^* &= \underset{\mathbf{P}}{\operatorname{argmin}}\ E_{Y|x}[L(y, F(\mathbf{x}; \mathbf{P}))] \\ &= \underset{\mathbf{P}}{\operatorname{argmin}}\ \mathbf{\Phi}(\mathbf{P}) \end{aligned}$$

And we denote our estimated function as $F^* = F(\mathbf{x}; \mathbf{P}^*)$

3.5.1 Numerical Optimization on Additive Models

Now, we restrict our attention to additive models. We define our additive as following:

$$F(\mathbf{x}; \{ \beta_m, \mathbf{a}_m \}_{m=1}^M) = \sum_{m=1}^{M} \beta_m h(\mathbf{x}; \mathbf{a}_m) \tag{15}$$

So, the $\mathbf{P}$ corresponds to the parameter set $\{ \beta_m, \mathbf{a}_m \}_{m=1}^M$ and $h(\mathbf{x}; \mathbf{a})$ is a simple model obtained by a weak learner. We will be using small regression trees as our weak learners. In that case, the parameters, $\mathbf{a}_m$, corresponds to split variables, split points and predictions at leaf (mean, median, etc. for regression trees). And the parameter β_m is the weight of the weak learner.

If we make an analogy to gradient descent, in which we make an update with a function's steepest direction to find the point where it is minimum:

$$x \coloneqq x - \alpha * f'(x)$$

where α is the learning data.

Or if we want to find the parameters of a function where it attains its minimum value, we make updates using a cost function $J(\theta)$:

$$\theta_i \coloneqq \theta_i - \alpha \frac{\partial J(\theta)}{\partial \theta_i}$$

With the same logic, in Gradient Boosting, we make updates to our additive model:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + f_m(\mathbf{x})$$

where $f_m(\mathbf{x}) = -\rho_m g_m(\mathbf{x})$ and

$$g_m(\mathbf{x}) = \left[\frac{\partial E_y[L(y, F(\mathbf{x})|\mathbf{x}]}{\partial F(\mathbf{x})} \right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$$

and the final solution will be:

$$F^*(\mathbf{x}) = \sum_{m=0}^{M} f_m(\mathbf{x})$$

where $f_0(x)$ is the initial guess.

We also find the optimum ρ_m as:

$$\rho_m = \underset{\rho}{\operatorname{argmin}}\ E_{y,\mathbf{x}} L(y, F_{m-1}(\mathbf{x}) - \rho g_m(\mathbf{x}))$$

3.5.2 Training

This method cannot be directly applied when we have limited data because we cannot calculate the expected loss directly. In this case, we make use of the training set that we have.

We will use the parameterized additive model in (15) and minimize expected loss estimated from the training set:

$$\{\ \beta_m, \mathbf{a}_m\ \} = \underset{\beta'_m, \mathbf{a}'_m}{\operatorname{argmin}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=0}^{M} \beta'_m h(\mathbf{x}_i; \mathbf{a}'_m)\right)$$

It's often hard to find all the parameters at one step. Instead of this, we use an iterative approach.

$$\{\ \beta_m, \mathbf{a}_m\ \} = \underset{\beta'_m, \mathbf{a}'_m}{\operatorname{argmin}} \sum_{i=1}^{N} L\left(y_i, F_{m-1}(\mathbf{x}_i) + \beta'_m h(\mathbf{x}_i; \mathbf{a}'_m)\right)$$

then we make update:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}; \mathbf{a}_m)$$

If you recall, this is the Forward Stagewise Additive Modelling technique that we talked about in the previous section.

In the case where we have finite data, g_m, the gradients are calculated for the training data instances $\{\mathbf{x}_i\}_{i=1}^N$:

$$g_m(\mathbf{x}_i) = \left[\frac{\partial L(y, F(\mathbf{x}_i)}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$$

But we cannot calculate the gradients directly for new data points other than the ones in the training set. And even if for training set, if we use the gradients directly,

the model would not be well generalized. Therefore, we need our model to learn a mapping from training data points to gradients in order to generalize to unseen data. To do that, we use a parameterized function and that is the $h(\mathbf{x}; \mathbf{a})$, as we mentioned and learn its parameters, $\mathbf{a}$, as given below:

$$\mathbf{a}_m = \underset{\mathbf{a}}{\operatorname{argmin}} \sum_{i=1}^{N} (-g_m(\mathbf{x}_i) - \beta h(\mathbf{x}_i; \mathbf{a})) \tag{16}$$

So, we fit the negative gradients, $-g_m$, to the parameterized model $h(\mathbf{x}; \mathbf{a})$ to learn a mapping from the obversations to its gradients. Negative gradients are also called "pseudo-responses", in the sense that, we try to learn a mapping to them even though they are not the real response values. And they are also called "pseudo-residuals" as well.

Therefore, we have a general algorithm that we will work for any differentible loss function. At each stage of the algorithm we learn a mapping from data points to gradients. This is analogous to the standard applications of gradient descent in machine learning where the parameters of a function is learned.

Algorithm 5: Gradient Boosting

Output: $F_M(\mathbf{x})$

$F_0(\mathbf{x}) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, c)$

for $m \leftarrow 1$ ***to*** M **do**

$\widetilde{y}_i = - \left[\frac{\partial L(y, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)} \right]_{F(\mathbf{x}_i) = F_{m-1}(\mathbf{x}_i)} \quad i = 1, 2, \ldots, N$

$\mathbf{a}_m = \underset{\mathbf{a}, \beta}{\operatorname{argmin}} \sum_{i=1}^{N} \left(\widetilde{y}_i - \beta h(\mathbf{x}_i; \mathbf{a}) \right)^2$

$\rho_m = \underset{\rho}{\operatorname{argmin}} \sum_{i=1}^{N} L\left(y_i, F_{m-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i; \mathbf{a}) \right)$

$F_m(\mathbf{x}) = F_{m-1}(x) + \rho_m h(\mathbf{x}_i; \mathbf{a}_m)$

end

return $F_M(\mathbf{x})$

3.5.3 Applications of Gradient Boosting

Let us derive some algorithms for common tasks such as regression and classification using the Gradient Boosting methodology that is presented in the previous section.

3.5.4 Least Squares Regression

We define the loss function for least squares regression as $L(y, F) = \frac{(y-F)^2}{2}$, the squared error. Pseudo-responses are derivative of this loss function. So, $\widetilde{y}$ is simply $y - F$. Our initials guess will be $F_0(\mathbf{x}) = \bar{y}$, namely the mean of the target values, because squared error is minimized at the mean. With these, we can build our least squares regression with Gradient Boosting algorithm.

Algorithm 6: Least Squares Regression

Output: $F_M(\mathbf{x})$
$F_0(\mathbf{x}) = \bar{y}$
for $m \leftarrow 1$ ***to*** M **do**
  $\widetilde{y_i} = y_i - F_{m-1}(\mathbf{x}_i)\ i = 1, 2, \ldots, N$
  $\{\ \mathbf{a}_m, \rho_m\ \} = \underset{\mathbf{a},\rho}{\operatorname{argmin}} \sum_{i=1}^{N} \left(\widetilde{y_i} - \rho h(\mathbf{x}_i; \mathbf{a})\right)^2$
  $F_m(\mathbf{x}) = F_{m-1}(x) + \rho_m h(\mathbf{x}_i; \mathbf{a}_m)$
end
return $F_M(\mathbf{x})$

3.5.5 Binary Classification

In the case of binary classification, we have negative binomial log-likelihood as the loss function:

$$L(y, F) = -(y \log p + (1 - y) \log(1 - p))$$

where $y \in \{-1, 1\}$. p is related to F through:

$$p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-F(\mathbf{x})}}$$

Pulling out F from this equation, we find:

$$F(\mathbf{x}) = \frac{1}{2} \log \left[\frac{P(y = 1|\mathbf{x})}{P(y = -1|\mathbf{x})} \right]$$

With some algebraic manipulation, we can write the same loss function using only y and F:

$$L(y, F) = \log(1 + \exp(-2yF))$$

Taking derivative with respect to F, we find the pseudo-response:

$$\widetilde{y_i} = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x}_i)} = \frac{2y_i}{1 + \exp(2y_i F_{m-1}(\mathbf{x}_i))} \tag{17}$$

where $i = 1, 2, \ldots, N$

We will use regression trees as our base learners to learn a mapping to these pseudo-responses. After building the regression tree, predictions in the leaves, R_{jm}, is the solution of this objectve:

$$\gamma_{jm} = \underset{\gamma}{\operatorname{argmin}} \sum_{\mathbf{x} \in R_{jm}} \log\left(1 + \exp(-2y_i(F_{m-1}(\mathbf{x}) + \gamma))\right) \tag{18}$$

We can't directly solve this equation. Instead, we will estimate it with a single Newton-Raphson step. For that, we need to find the first and second derivative of:

$$H(\gamma; R_{jm}) = \sum_{\mathbf{x} \in R_{jm}} \log\left(1 + \exp(-2y_i(F_{m-1}(\mathbf{x}) + \gamma))\right) \tag{19}$$

1-step Newton-Raphson approximation of gamma with initial value 0 is:

$$\begin{aligned}
\gamma &= \gamma_0 - \frac{H'(\gamma_0)}{H''(\gamma_0)} \\
&= 0 - \frac{H'(0)}{H''(0)} \\
&= -\frac{H'(0)}{H''(0)}
\end{aligned}$$

First derivative of H is:

$$H'(\gamma) = \sum_{\mathbf{x} \in R_{jm}} \frac{-2y_i}{1 + \exp(2y_i(F_{m-1}(\mathbf{x}_i) + \gamma))} \tag{20}$$

And at $\gamma = 0$:

$$H'(0) = \sum_{\mathbf{x} \in R_{jm}} \frac{-2y_i}{1 + \exp(2y_i F_{m-1}(\mathbf{x}_i))}$$

We can see this equation is equal to negative of $\widetilde{y}$ (see equation (17)). So:

$$H'(\gamma) = -\sum_{\mathbf{x} \in R_{jm}} \widetilde{y}_i \tag{21}$$

Second derivative of H can be found by taking derivative of (19) with respect to γ again:

$$\begin{aligned}
H''(\gamma) &= \sum_{\mathbf{x} \in R_{jm}} \frac{d}{d\gamma} H'(\gamma) \\
&= \sum_{\mathbf{x} \in R_{jm}} \frac{(-2y_i)2y_i \exp(2y_i F_{m-1}(\mathbf{x}))}{(1 + \exp(2y_i(F_{m-1}(\mathbf{x}) + \gamma)))^2} \\
&= \sum_{\mathbf{x} \in R_{jm}} -\widetilde{y} \,.\, \widetilde{y} \,.\, exp(2y_i F_{m-1}(\mathbf{x})) \\
&= \sum_{\mathbf{x} \in R_{jm}} -\widetilde{y} \,.\, \widetilde{y} \,.\, \frac{(2y - \widetilde{y})}{\widetilde{y}} \\
&= \sum_{\mathbf{x} \in R_{jm}} \widetilde{y}(\widetilde{y} - 2y)
\end{aligned}$$

We can simplify this a little bit further. We can see that, from the equation (17), y and $\widetilde{y}$ always has the same sign. So, the product $y \,.\, \widetilde{y}$ equals to $|\widetilde{y}|$.

Therefore, the second derivative equals to:

$$H''(\gamma) = \sum_{\mathbf{x} \in R_{jm}} |\widetilde{y}| \; (|\widetilde{y}| - 2) \tag{22}$$

Being calculated the first and second derivatives, we can find our 1-step Newton-Raphson approximation of γ:

$$\gamma_{jm} = -\frac{H'(0)}{H''(0)}$$
$$= -\frac{\sum_{\mathbf{x}\in R_{jm}} \widetilde{y}}{\sum_{\mathbf{x}\in R_{jm}} |\widetilde{y}| \ (|\widetilde{y}| - 2)}$$
$$\gamma_{jm} = \frac{\sum_{\mathbf{x}\in R_{jm}} \widetilde{y}}{\sum_{\mathbf{x}\in R_{jm}} |\widetilde{y}| \ (2 - |\widetilde{y}|)} \tag{23}$$

By using (23), we can label the leaves of the decision tree that was built in mth iteration.

Finally, we must derive an initial prediction, $F_0(\mathbf{x})$. One can easily show that the negative binomial log-likelihood is minimized at:

$$F_0(\mathbf{x}) = \frac{1}{2} \log \left(\frac{\sum_{i=1}^{N} 1(y_i = 1)}{\sum_{i=1}^{N} 1(y_i = -1)} \right) \tag{24}$$

Notice that we used $\frac{1}{N} \sum_{i=1}^{N} 1(y_i = 1)$ as an estimate for $P(y = 1|\mathbf{x})$. Similarly $\frac{1}{N} \sum_{i=1}^{N} 1(y_i = -1)$ is an estimate for $P(y = -1|\mathbf{x})$. By using these estimates, we came up with (24).

Note that the F_0 is refers to a constant value that minimizes cost

Equation above becames a lot clear if we used $\bar{y} = \frac{1}{N} \sum_{i=1}^{N} y_i$:

$$F_0(\mathbf{x}) = \frac{1}{2} \log \frac{1 + \bar{y}}{1 - \bar{y}} \tag{25}$$

Algorithm 7: Binary Classification With Gradient Boosting

Output: $F_M(\mathbf{x})$

$F_0(\mathbf{x}) = \frac{1}{2}\log\frac{1+\bar{y}}{1-\bar{y}}$

for $m \leftarrow 1$ ***to*** M **do**

$\widetilde{y}_i = \frac{2y_i}{1+\exp(2y_i F_{m-1}(\mathbf{x}_i))}\ i = 1, 2, \ldots, N$

$\{R_{jm}\}_{j=1}^{J} = DecisionTree\ with\ J - leaves$

$$\{\ \gamma_{jm}\ \} = \frac{\sum\limits_{\mathbf{x}\in R_{jm}} \widetilde{y}}{\sum\limits_{\mathbf{x}\in R_{jm}} |\widetilde{y}|\ (2 - |\widetilde{y}|)}\ j = 1, 2, \ldots, J$$

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

end

return $F_M(\mathbf{x})$

Following the same logic, any learning task with any differentiable loss function L and base learner h, can be done with Gradient Boosting algorithm presented in Algorithm 5.

3.5.6 Regularization

Usually, we don't want our learning algorithm to "overlearn" our training set. Instead, we want them to generalize and have high performance on unseen data as well as training data. If a learning algorithm performs almost perfect on training data but perform poorly on a separate validation set, then that algorithm is said to be overfit. There are techniques to overcome this problem of overfitting. They are called regularization techniques. Regularization techniques differs from algorithm to algorithm.

For example, in gradient descent, we regularize our model by tuning the learning rate parameter or number of iterations. Learning rate is the shrinkage parameter applied on the gradients of the cost function.

In Gradient Boosting, we train M base learner to learn a mapping to gradients. Then update our model with these gradients. So, one natural regularization technique is tuning the M parameter. Other one is bringing a new learning rate parameter ν, to the model. We can modify our model update equation with this learning rate parameter as the following:

$$F_m(\mathbf{x}) = F_{m-1}(x) + \nu\ .\ \rho_m h(\mathbf{x}_i; \mathbf{a}_m)$$

By using a learning rate, we reduce the influence of a single base learner to leave some room for other base learners to improve the model.

There are other effective regularization techniques that are used by popular Gradient Boosting libraries such as XGBoost, LightGBM, Catboost, etc. Popular ones includes maximum features to use at each iteration (this is similar to Random Forest), subsampling the training set at each iteration and maximum depth of each tree, etc.

In the next section, we will discuss these popular algorithms that uses the Gradient Boosting concept.

3.6 XGBoost

In recent years, several Gradient Boosting algorithms has been developed. With these algorithms, Gradient Boosting became much more scalable and computationally efficient. In Gradient Boosted Decision Trees, most computationally expensive part is the tree building process. For each non-terminal node, a split criterion must be found by looking at all possible splits of each feature. This process is really slows down the training of Gradient Boosted Decision Trees. If we have lots of features and lots of data, even one step of Gradient Boosting takes an unreasonable amount of time. Recently developed algorithms address this problem and gives efficient solutions.

One popular Gradient Boosting franework is XGBoost. It was initially developed in 2014. It uses a modified cost function that has an additional regularization term which penalizes the complexity of the trees. Besides that, it addresses computational problems that arise when using Gradient Boosting in large data sets by proposing several solutions.

3.6.1 Regularized Cost Function

As we seen on the previous chapter, given a loss function, L, we define our objective (cost function) as:

$$\sum_{i=1}^{N} L(y_i, F_m(\mathbf{x}_i))$$

XGBoost proposes a modified version of this objective that takes regularization into account:

$$\mathcal{L} = \sum_{i=1}^{N} L(y_i, F_M(\mathbf{x}_i)) + \sum_{k=1}^{M} \Omega(f_k) \tag{26}$$

where $\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2$. T is the number of leaves in the tree f and w_j is the score in jth leaf. γ and λ are regularization parameters.

This objective minimizes the loss of the final model F_M with M trees. However, in practice, it's impossible to find M trees in only one step. We need a greedy approach that adds the trees in an iterative fashion. So, at iteration m, we need to solve the objective:

$$\mathcal{L}^{(m)} = \sum_{i=1}^{N} L(y_i, F_{m-1}(\mathbf{x}_i) + f_m(\mathbf{x}_i)) + \Omega(f_m) \tag{27}$$

Assuming that L is a twice differentiable function, we approximate the L function by second order Taylor polynomial.

$$\mathcal{L}^{(m)} = \sum_{i=1}^{N} \left[L(y_i, F_{m-1}(\mathbf{x}_i)) + g(\mathbf{x}_i) f_m(\mathbf{x}_i) + \frac{1}{2} h(\mathbf{x}_i) f_m^2(\mathbf{x}_i) \right] + \Omega(f_m) \tag{28}$$

where

$$g(\mathbf{x}) = \frac{\partial L(y_i, F_{m-1}(\mathbf{x}))}{\partial F_{m-1}(\mathbf{x})}$$

and

$$h(\mathbf{x}) = \frac{\partial L(y_i, F_{m-1}(\mathbf{x}))^2}{\partial^2 F_{m-1}(\mathbf{x})}$$

This equation can be simplified by removing the constant terms.

$$\tilde{\mathcal{L}}^{(m)} = \sum_{i=1}^{N} \left[g(\mathbf{x}_i) f_m(\mathbf{x}_i) + \frac{1}{2} h(\mathbf{x}_i) f_m^2(\mathbf{x}_i) \right] + \Omega(f_m)$$

$$\tilde{\mathcal{L}}^{(m)} = \sum_{i=1}^{N} \left[g(\mathbf{x}_i) f_m(\mathbf{x}_i) + \frac{1}{2} h(\mathbf{x}_i) f_m^2(\mathbf{x}_i) \right] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2$$

To simplify this equation further, we can group the summations to be based on the points in the same leaf. Recall that we can define the f_m as:

$$f_m(\mathbf{x}) = \sum_{j=1}^{T} w_j 1(\mathbf{x} \in R_j)$$

Using that definition of f_t, we can simplify our objective:

$$\begin{aligned}\tilde{\mathcal{L}}^{(m)} &= \sum_{j=1}^{T}\left[\sum_{\mathbf{x}_i\in R_j} g(\mathbf{x}_i)w_j + \frac{1}{2}\sum_{\mathbf{x}_i\in R_j} h(\mathbf{x}_i)w_j^2\right] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} {w_j}^2 \\ &= \sum_{j=1}^{T}\left[\sum_{\mathbf{x}_i\in R_j} g(\mathbf{x}_i)w_j + \frac{1}{2}\left(\sum_{\mathbf{x}_i\in R_j} h(\mathbf{x}_i) + \lambda\right) w_j^2\right] + \gamma T\end{aligned}$$

We need to find w_j's that need minimize this equation. Taking derivative with respect to w_j, we can find the optimum w_j:

$$w_j^* = -\frac{\sum\limits_{\mathbf{x}_i\in R_j} g(\mathbf{x}_i)}{\sum\limits_{\mathbf{x}_i\in R_j} h(\mathbf{x}_i) + \lambda} \tag{29}$$

Using these w_j's, we can find the optimum cost value:

$$\tilde{\mathcal{L}}^{(m)^*} = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum\limits_{\mathbf{x}_i\in R_j} g(\mathbf{x}_i)\right)^2}{\sum\limits_{\mathbf{x}_i\in R_j} h(\mathbf{x}_i) + \lambda} + \gamma T \tag{30}$$

We can use this equation as a scoring function when deciding on split criteria. As discussed in Decision Trees chapter, when we looking for the best split across all feature space, we choose the feature and the split point that gives the most reduction in our impurity. Likewise, XGBoost chooses the split crtierion that results in most reduction in the cost (28).

If a region R is splitted into two regions R_L and R_R, then the reduction in the cost after the split is given by:

$$\tilde{\mathcal{L}}^{(m)^*} = \frac{1}{2}\left[\frac{(\sum\limits_{\mathbf{x}_i\in R_L} g(\mathbf{x}_i))^2}{\sum\limits_{\mathbf{x}_i\in R_L} h(\mathbf{x}_i) + \lambda} + \frac{(\sum\limits_{\mathbf{x}_i\in R_R} g(\mathbf{x}_i))^2}{\sum\limits_{\mathbf{x}_i\in R_R} h(\mathbf{x}_i) + \lambda} - \frac{(\sum\limits_{\mathbf{x}_i\in R} g(\mathbf{x}_i))^2}{\sum\limits_{\mathbf{x}_i\in R} h(\mathbf{x}_i) + \lambda}\right] - \gamma \tag{31}$$

3.6.2 Split Finding

As we discussed in the previous chapter, split finding is arguably the most computationally expensive part of the Decision Tree algorithm. To find the best split for a specific node, we need to iterate over all features and sort their values, and search for the values that gives the best split according to some impurity measure like gini index, cross entropy or cost reduction like (31). This algorithm is called the Exact Greedy Algorithm. In settings where we have millions or billions of data points, this solution becomes infeasible.

Instead, XGBoost proposes several algorithms for avoiding the disadvantages of Exact Greedy Algorithm for split finding.

3.6.3 Approximate Algorithm

Exact Greedy algorithm needs to enumerate over all values of a feature in one step and it needs to repeat that for all features. To avoid the computational cost of this process, XGBoost proposes an algorithm called Approximate Algorithm. What it does is, instead of looking at every value of a feature, it finds l percentile of a feature, based on its values in the training set. Then, it iterates over these l different percentiles to find the best possible split amongst these percentiles, instead of looking at $N_k \approx N$ different values of a feature. As the Exact Greedy algorithm, Approximate Algorithm does this step for every features.

3.6.4 Sparsity-aware Split Finding

There can be a lot of sparse features in large data sets. This sparsity in the data slows down the split finding process. XGBoost proposes a Sparsity-aware Split Finding algorithm. It gives a default direction (left or right) to which a way the missing values in the feature to go while splitting the node. It finds the best direction from the data by looking at only the non-missing values and calculating the cost reduction in each direction.

There are also low level optimizations XGBoost performs that speeds up the tree learning process. Its paper explains them in detail.

4 Resources

In this section, I'll give the resources I used while preparing this article.

You can download the Wisconsin Breast Cancer Data Set from here: `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`.

Ethem Alpaydın's Introduction to Machine Learning book has a great Decision Tree chapter that gives a solid introduction. Stanford University's CS229 course has useful material in various topics including Decision Trees, Random Forest, Boosting, etc.

A couple of detailed material about CART can be found from these links: 1 2

Elements Of Statistical Learning book by Trevor Hastie, Robert Tibshirani, Jerome H. Friedman is a great book that covers lots of topics in Machine Learning. It has separate dedicated chapters about the topics covered in this article and it gives clear, detailed explanations about these topics.

Jerome H. Friedman's Greedy Function Approximatiın: Gradient Boosting Machine paper gives a thorough description of Gradient Boosting algorithm and derives several algorithms using Gradient Boosting.

XGBoost: A Scalable Tree Boosting System paper by Tianqi Chen and Carlos Guestrin describes the XGBoost framework. Interested readers are encouraged to read it if they want to learn about what optimizations XGBoost does more deeply.