Survey of Boosting Methods

Caijun Qin
cqin@ufl.edu
University of Florida

1. Introduction

Boosting denotes a general group of ensemble-learning algorithms that collectively make a final prediction using additive mechanism (regression) or an additional voting mechanism (classification) by individual models called weak learners. A weak learner is an instance of the base estimator that only uses one or a few features form the data for regression or classification, hence "weak". In boosting, typically all weak learners are based on the same type of model i.e. the base estimator, such as decision tree. This survey provides an conceptual overview and simplified mathematical explanation of algorithms for several influential boosting methods.

2. Boosting Methods

2.1 AdaBoost

Short for "Adaptive Boosting", AdaBoost was conceived by Yoav Freund and Robert Schapire and delivered as part of a written lecture (Freund and Schapire, 1999). The base estimator of AdaBoost is simply a stump, which is a decision tree with a single split from the root node. This boosting method presents a robust way of making predictions on data with highly nonlinear boundaries with two sets of weighting in its algorithm. Although adaptation for regression exists, the process of building an AdaBoost model is explained under the context of classification in the next section. To lessen any potential ambiguity in notation, let $H(x_i)$ denote the final prediction of the model for the i^{th} observation of some dataset. Let $t = 1,2,\dots,m$ refer to an arbitrary (or rather t^{th}) stump and iteration of AdaBoost composed of m stumps.

Each observation x_i holds a weight throughout the process of appending stumps. Regardless of which iteration, the observation weights are always normalized before stump creation. Initially, all weights are equal. Since the sum of all weights is constrained to 1, initial weights are $w_1 = w_2 = \cdots = w_n = 1/N$. The weight of an observation reflects how frequently that observation is misclassified as new stumps become appended and therefore are continuously updated during each iteration. Each update to a weight consists of a scaling of the same weight from the previous iteration. After a stump is created, an error must be computed for that stump, $err_t = \frac{\Sigma_{i=1}^{N} w_i I(h_t(x_i) \neq c_i)}{\Sigma_{i=1}^{N} w_i}$. This is simply the weighted proportion of observations that are incorrectly predicted.

A separate, unrelated weight is then computed for the newly created stump on iteration t, $\alpha_t = \ln \frac{1-err_t}{err_t}$, which reflects how strongly the stump correctly or incorrectly classifies its input dataset. $\alpha_t > 0$ when a majority are correctly classified ($err_t < 0.5$), and $\alpha_t < 0$ when a majority are misclassified ($err_t > 0.5$). On the very unlikely chance that $err_t = 0.5$, the stump is no better than random guessing (for binary classification) and $\alpha_t = 0$, deeming the stump insignificant.

The scaling factor multiplied to the current weight is then computed by $\exp\left[\alpha_t I_{\{-1,1\}}(h_t(x_i) \neq c_i)\right]$. Note that the indicator function *I* outputs either 1 or -1 rather than 1 or 0. In summary, the current weight for x_i is scaled upwards for incorrect classification by an accurate stump or correct classification by an inaccurate stump. Otherwise, the weight is scaled downwards. Therefore, observations difficult for classification becomes more emphasized. All observation weights are normalized after updating.

In each new iteration, the input data used to construct the stump comes from a resampling of *N* observations with replacement from the input data used to generate the previous stump. Relatively highly-weighted observations would have greater probability of being chosen, possibly more than once, for the input data in the next iteration. This process underscores one of AdaBoost's main strategies, which is to use proportionally more difficult observations to classify to design new stumps until correct classification for such observations can be seen.

Algorithm 2 : AdaBoost.M1

1 **Input :** Dataset $D = \{(a_1, c_1), (a_2, c_2), \ldots, (a_N, c_N)\}$,
Base learner L and
Number of learning iteration T

2 Initialize equal weight to all training samples $w_i = 1/N$, $i = 1,2,\ldots, N$

3 For t=1 to T :

(a) Train a base learner h_t from D using D_t to training sample using w_i.
$h_t = L(D, D_t)$

(b) Compute error of h_t as
$$err_t = \frac{\sum_{i=1}^{N} w_i I(h_t(a_i) \neq c_i)}{\sum_{i=1}^{N} w_i}$$

(c) Compute the weight of h_t as
$$\alpha_t = \log\left(\frac{1 - err_t}{err_t}\right)$$

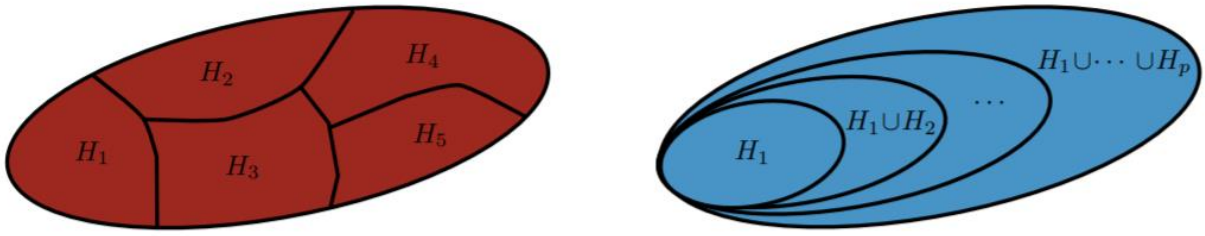
(d) Set $w_i \leftarrow w_i . \exp[\alpha_t I(h_t(a_i) \neq c_i)]$

4 **Output :** $H(a) = \text{sign} \sum_{t=1}^{T} \alpha_t h_t(a)$

Figure 1. AdaBoost.M1 algorithm as pseudo code.

2.2 DeepBoost

Developed by Google Research and the Courant Institute at New York University, DeepBoost is relatively less popular and more recent boosting method than AdaBoost. The main idea behind DeepBoost entails re-organizing the set of weak learners from some space *H*, the set of all possible learners creatable from s dataset, into a hierarchical nesting ordered by model complexity. Complexity in the context of a decision tree as the base estimator typically constitutes tree depth. In Figure 2, H_k corresponds to the set of all weak learners with a depth of *k*.



Figures 2-3. Re-organization of weak learners by complexity, e.g. tree depth, to form a nested set view of weak learners for visual interpretation of DeepBoost.

In Figure 3, the nested structure of weak learners varying in complexity can be expressed as $F = conv\left(\cup_{k=1}^{p} H_k\right)$, where H_k refers to the group of weak learners with complexity, in this case tree depth, of *k*. There are many error functions utilized in DeepBoost. For the simple case of binary classification, $R(f) = E_{(x,y)\sim D}\left[1_{yf(x)\leq 0}\right]$ describes the error, which is simply the expected value of the distribution of true values $y \in \{-1,1\}$ with distribution D. One loss function that can be computed on any iteration during the appendage of weak learners is $L(x_i) = \frac{1}{2}\Big[1 - E_{i\sim D_t}[y_i h_i(x_i)]\Big]$, where D_t refers to distribution over sample on the t^{th} iteration of building the ensemble of weak learners.

2.3 Gradient Boost

Gradient boosting presents an additive model that multiples all weak learner predictions after the first one by a constant learning rate before summation into a final prediction. The additive nature, without any assignment or sign function as in AdaBoost, gives an edge for gradient boost in regression tasks. Gradient boost also differs from AdaBoost in base estimator. While AdaBoost restricts the base estimator to a stump, each weak learner in gradient boost and related boosting methods, including xgBoost, typically use decision trees with a depth of 3 to 5 levels. Furthermore, appending more weak learners constitutes gradient boost 's strategy of decreasing the total residual between predicted values and true values (Friedman, 1999). On iteration *m*, a new decision tree is fitted from pseudo-residuals from the previous or *m-1*th iteration, with a pseudo-residual pertaining to x_i computed as $r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$. This in effect uses the loss gradient to determine which direction the gap, hence residual, between the predicted value given x_i and true value y_i is currently oriented when using only the previous *m-1* weak learners as part of boosting. Negating this value would theoretically decrease the residual.

The first weak learner is a constant value that satisfies $\arg\min_{\gamma} \Sigma_{i=1}^{N} L(y_i, \gamma)$. Popular for regression, we can specify *MSE* as the loss function *L*. To solve for γ, setting the first-order derivative to zero (0) would be an efficient solution.

$$\arg\min_{\gamma} \Sigma_{i=1}^{N} L(y_i, \gamma) \Rightarrow \arg\min_{\gamma} \frac{1}{\mathrm{N}} \Sigma_{i=1}^{N} (y_i - \gamma)^2 \Rightarrow \arg\min_{\gamma} \frac{1}{2} \Sigma_{i=1}^{N} (y_i - \gamma)^2$$
$$\Rightarrow \frac{\mathrm{d\,MSE}}{\mathrm{d}\,\gamma} = 0 \Rightarrow \Sigma_{i=1}^{N} (y_i - \gamma) = 0 \Rightarrow \gamma N = \Sigma_{i=1}^{N} y_i \Rightarrow \gamma = \frac{1}{N} \Sigma_{i=1}^{N} y_i = \bar{y}$$

Therefore, having a gradient boost model with $m = 1$ weak learner would always output the mean regardless of feature values from the input. For each iteration afterwards, we compute γ again but with $\gamma_{jm} = \arg\min_{\gamma} \Sigma_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$, where R_{jm} refers to a specific terminal node indexed by j in the m^{th} decision tree. By this equation, it is clear that γ represents the predicted value associated with a specific terminal node which minimizes the total sum of residuals pertaining to the constituent predicted values from the $m\text{-}1^{th}$ decision tree in that same terminal node. The first decision tree is simply a special case, as the weak learner is a single root with no splits, and the mean $\bar{y}$ would be the best prediction. After the first weak learner, the RHS of $\hat{y}_i = f_{m-1}(x_i)$ can be seen in the computation of γ_{jm} to account for the predicted value by a terminal node in the most recent growth of the gradient boost model.

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

Figure 5. Pseudocode for gradient boost algorithm.

The metric utilized for loss are different depending on whether gradient boost is performing a regression or classification task. For regression, *mean-squared error (MSE)* is a popular choice and is computed as $\frac{1}{N}\Sigma_{i=1}^{N}(y_i - \hat{y}_i)^2$. However, classification converts probabilities of each class in y to log odds as numerical inputs, and *log loss* is preferred as the loss function. Log loss function for the general multiclass case and binary case are given below, respectively.

$LogLoss = -\frac{1}{N}\Sigma_{i=1}^{N}\Sigma_{j=1}^{M} y_{ij} \ln P(y_i \in c_j)$ for N observations distributed among M classes

$LogLoss = -\frac{1}{N}\Sigma_{i=1}^{N}[y_i \ln P(y_i) + (1 - y_i) \ln P(1 - y_i)]$

2.4 xgBoost

Short for "Extreme" gradient boost, the conception of xgBoost proved to be a pivotal moment in boosting. xgBoost introduced numerous optimizations and accuracy improvement strategies over vanilla gradient boosting (Chen & Guestrin, 2016). Rather than attempting to minimize residuals after each appended weak learner as in gradient boost, xgBoost reverts direction by attempting to maximize information gain to choose thresholds for splits when building a weak learner. Just like gradient boost, the base estimator is also a decision tree with preferably 3 to 5 levels in depth. Per iteration of appending weak learners, choosing the threshold to split existing nodes in the learner constitutes a core step in xgBoost algorithm. The following formulas are utilized.

For Regression

$Similarity\ Score = \frac{\left(\Sigma_{i=1}^{N} e_i\right)^2}{N+\lambda}$

$Output\ Value = \frac{\Sigma_{i=1}^{N} e_i}{N+\lambda}$

For Classification

$Similarity\ Score = \frac{\left(\Sigma_{i=1}^{N} e_i\right)^2}{\Sigma_{i=1}^{N} P(f_{m-1}(x_i))(1-P(f_{m-1}(x_i))+\lambda}$

$Output\ Value = \frac{\Sigma_{i=1}^{N} e_i}{\Sigma_{i=1}^{N} P(f_{m-1}(x_i))(1-P(f_{m-1}(x_i))+\lambda}$

Note that for classification, output values, whether predicted or actual, are log odds. For classification, each residual would be the difference between predicted and actual log odds. The similarity score quantifies how closely clustered on the same side of a threshold for observations within a specific node. The threshold is often centered to zero. The gain in importance, or rather gain in similarity, is simply the difference between combined similarity scores of child nodes and the parent node, symbolically $Gain = Gain_L + Gain_R - Gain_{parent}$. For any numerical feature, a potential threshold for node splitting exists between every pair of consecutive values when observations become sorted by that specific feature. The threshold leading to a split with largest gain in similarity would be chosen. This process continues for newly made terminal nodes to continue deepening the decision tree.

Note that a penalty constant λ can be explicitly set. A positive λ decreases similarity score than without the penalty, which aids against overfitting at the split level. If a node to be split originally contains very few observations, such as only two, the even fewer observations split into each child node would allow sparsity of the observations to inordinately contribute to the similarity score. As an extreme example, a child node with only one observation would of course be closely clustered with itself. Therefore, λ can be viewed to "smooth" against the effects of sparse observations during splits.

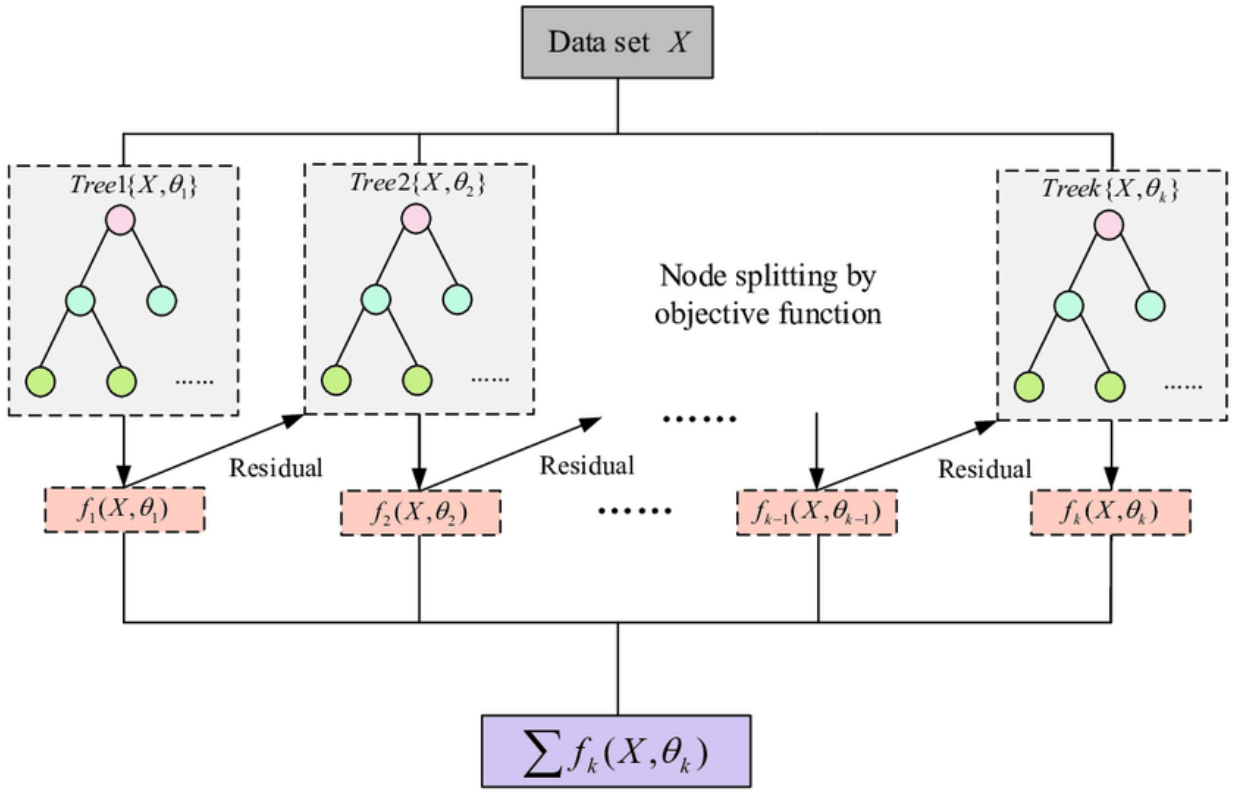


Figure 6. Node splitting iterative process.

xgBoost provides many opportunities to curb overfitting or otherwise limit the growth of the boosting model both at the decision tree level and the appendage of new weak learners. A nonnegative value γ can be set as a minimum threshold for gain in similarity. When attempting any new split with a gain of similarity falling less than γ, the split becomes aborted. This process forms the basis of pruning for xgBoost. Even without explicitly setting γ, a value of zero would still work as similarity gain decreases as a decision tree grows in height. On the other extreme, xgBoost can specify a *cover*, which is the minimum number of splits per decision tree.

3. References

Freund, Y., & Schapire, R. E. (1999). *A Short Introduction to Boosting*. http://arxiv.org/abs/1508.01136

Cortes, C., Mehrmyar, M., & Usyed, U. (2014). *Deep Boosting*. *32*, 1–9. papers://d471b97a-e92c-44c2-8562-4efc271c8c1b/Paper/p613

Friedman, J. H. (1999). *A Greedy Function Approximation: A Gradient Boosting Machine*.

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, *13-17-Augu*, 785–794. https://doi.org/10.1145/2939672.2939785