How to Build a Tree?

Three elements:

1. Where to **split**?
2. When to **stop**?
3. How to **predict** at each leaf node?

- A split is denoted by (j, s): split the data into two parts based on "var $j <$ value s or not".
- For each split, define a **Goodness of split criterion** $\Phi(j, s)$, e.g., deduction of RSS for regression trees.

Start with the root. Try all possible variables $j = 1 : p$ and all possible split values (for each variable j, sort the n values from the n samples, and choose s to be a middle point of two adjacent values, so at most $(n - 1)$ possible values for s), and pick the best split, i.e., the split with the best Φ value. Now, data are divided into the left node and right node. Repeat this procedure in each node.

Goodness of Split $\Phi(j, s)$

For **Classification tree**, we check the gain of an **impurity measure**

$$\Phi(j,s) = i(t) - \left[p_R \cdot i(t_R) + p_L \cdot i(t_L)\right]$$

where

$$\begin{aligned} i(t) &= I(\hat{p}_t(1), \ldots, \hat{p}_t(K)) \\ \hat{p}_t(j) &= \text{frequency of class } j \text{ at node } t \\ I(\cdots) &= \text{an impurity measure.} \end{aligned}$$

Pick a split (j, s) that leads to a large reduction of impurity measure.

An **impurity measure** is a function $I(p_1, \ldots, p_K)$ where $p_j \geq 0$ and $\sum_j p_j = 1$ with properties

1. maximum occurs at $(1/K, \ldots, 1/K)$ (the most impure node);
2. minimum occurs at $p_j = 1$ (the purest node)
3. $I(\cdots)$ is symmetric function of $p_1, \ldots, p_K$, i.e., permutation of p_j does not affect ϕ.

Ideally we want the impurity measure to be small for each node.

Impurity Measures

- **Misclassification** rate : $1 - \max_j p_j$
- **Entropy** (deviance): $-\sum_{j=1}^{K} p_j \log p_j$
- **Gini index** :

$$\sum_{j=1}^{K} p_j(1-p_j) = 1 - \sum_j p_j^2$$

Specially when $K = 2$, we have

$$\begin{aligned} \text{Misclassification} &: \min(p, 1-p) \\ \text{Entropy} &: p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p} \\ \text{Gini index} &: 2p(1-p) \end{aligned}$$

The latter two are strictly **concave**, consequently the corresponding goodness-of-split measure Φ is always positive (unless the class frequency of the left node and the one of the right node are exactly the same), therefore they are often used in growing trees.

Consider a binary classification problem. Evaluate the following split:

$$(20, 5) \Longrightarrow (10, 0) \ + \ (10, 5).$$

- A node with 20 samples from “class 1" and 5 samples from “class 0" is split into two,
- left node has 10 samples from “class 1" and 0 samples from “class 0" and
- right node has 10 samples from “class 1" and 5 samples from “class 0".

- $\Phi(j, s)$ based on misclassification rate

$$\frac{5}{25} - \frac{10}{25} \cdot \frac{0}{10} - \frac{15}{25} \cdot \frac{5}{15} = 0$$

- $\Phi(j, s)$ based on entropy

$$\left[\frac{5}{25} \log \frac{25}{5} + \frac{20}{25} \log \frac{25}{20}\right] - \frac{10}{25} \cdot 0$$
$$- \frac{15}{25} \cdot \left[\frac{5}{15} \log \frac{15}{5} + \frac{10}{15} \log \frac{15}{10}\right] > 0$$

This split is regarded as zero gain if using Misclassification, but positive gain if using **Entropy or Gini** (which favor splits that lead to **pure nodes**).

Boosting Overview

- **AdaBoost**
  What exactly does it do? The resulting classifier will always have a good prediction accuracy?

- **Forward stage-wise optimization for fitting an additive model**
  AdaBoost is a special case of this framework with Exponential loss for classification. Similarly we can develop Boosting algorithms for regression/classification with other loss functions.

AdaBoost

Consider a binary classification problem with $y = \pm 1$. classifier

$$g : x \longrightarrow \{-1, 1\}.$$

Here g is a **weak classifier**, i.e., its performance is just slightly better than random guessing. In fact, it's okay that g is even worse than random guessing. Then you'll see that boosting automatically uses $-g(x)$.

Aim : use a combination of weak classifiers to improve the performance.

- Sequentially modify the weights on the training data $\{w_i\}_{i=1}^n$;
- Sequentially pick classifiers $g_t(x)$;[a]
- Output the weighted version

$$G(x) = \text{sign}\Big(\sum_{t=1}^{T} \alpha_t g_t(x)\Big).$$

[a] The algorithm still works if $g_t(x)$'s are chosen randomly.

The Algorithm

1. Initialize the weights $w_i^{(1)} = 1/n$, $i = 1, 2, \ldots, n$.
2. For $t = 1$ to T:
   (a) Fit a classifier $g_t(x)$;
   (b) Compute the training error wrt weights $w_i^{(t)}$'s

$$\epsilon_t = \sum_i w_i^{(t)} I\big(y_i \neq g_t(x_i)\big)$$

   (c) Compute $\alpha_t = \frac{1}{2} \log \frac{1-\epsilon_t}{\epsilon_t}$. Note $\alpha_t < 0$ if $\epsilon_t > 1/2$.
   (d) Update weights

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp[-\alpha_t y_i g_t(x_i)]}{Z_t},$$

   where Z_t is the normalizing constant to ensure that $\sum_i w_i^{(t+1)} = 1$.
3. Output $G_T(x) = \text{sign}\big(\sum_{t=1}^T \alpha_t g_t(x)\big)$.

Proof

Show that the Training Error (measured by mis-classification rate) will go to 0 (not necessarily monotonically) when $T \to \infty$.

$$
\begin{aligned}
\text{Training-Err}(G_T) &= \sum_i \frac{1}{n} I\Big(y_i \neq \text{sign}\big(\sum_{t=1}^{T} \alpha_t g_t(x_i)\big)\Big) \\
&= \sum_i \frac{1}{n} I\Big(\sum_{t=1}^{T} y_i \alpha_t g_t(x_i) < 0\Big) \\
&\leq \sum_i \frac{1}{n} \exp\Big(-\sum_{t=1}^{T} \alpha_t y_i g_t(x_i)\Big) \quad I(z<0) < e^{-z}, z \in \mathbb{R} \\
&\leq \prod_{t=1}^{T} Z_t
\end{aligned}
$$

$$
\begin{aligned}
& \sum_{i=1}^{n} \frac{1}{n} \exp\Big(- \sum_{t=1}^{T} \alpha_t y_i g_t(x_i) \Big) \\
= & \sum_{i} \frac{1}{n} \prod_{t=1}^{T} \exp\Big(- \alpha_t y_i g_t(x_i) \Big) \\
= & \sum_{i} w_i^{(1)} \prod_{t=1}^{T} \frac{w_i^{(t+1)}}{w_i^{(t)}} Z_t \\
= & \sum_{i} w_i^{(1)} \frac{w_i^{(2)}}{w_i^{(1)}} \cdots \frac{w_i^{(T)}}{w_i^{(T-1)}} \frac{w_i^{(T+1)}}{w_i^{(T)}} \Big(\prod_{t=1}^{T} Z_t \Big) \\
= & \Big(\prod_{t=1}^{T} Z_t \Big) \sum_{i} w_i^{(T+1)} \\
= & \prod_{t=1}^{T} Z_t,
\end{aligned}
$$

which decreases with T if $\epsilon_t < 1/2$, since

$$
\begin{aligned}
Z_t &= \sum_{i} w_i^{(t)} \exp\big(- \alpha_t y_i g_t(x_i) \big) \\
&= \sum_{i: y_i g_t(x_i) = 1} w_i^{(t)} \exp\big(- \alpha_t \big) + \\
&\quad \sum_{i: y_i g_t(x_i) = -1} w_i^{(t)} \exp\big(\alpha_t \big) \\
&= (1 - \epsilon_t) \exp\big(- \alpha_t \big) + \epsilon_t \exp\big(\alpha_t \big) \\
&= (1 - \epsilon_t) \sqrt{\frac{\epsilon_t}{1 - \epsilon_t}} + \epsilon_t \sqrt{\frac{1 - \epsilon_t}{\epsilon_t}} \\
&= 2 \sqrt{\epsilon_t (1 - \epsilon_t)} \\
&< 1
\end{aligned}
$$

- We can use a classifier $g_t(x)$ whose error rate $\epsilon_t > 1/2$ (i.e., worse than random-guessing).
  Then $\alpha_t < 0$, and Adaboost basically uses $-g_t(x)$.
- AdaBoost combines weak classifiers to reduce the 0/1 training error (or more specifically, reduce an upper bound of the training error). The training error of the combined classifier G_T (from Adaboost) is **not** monotonically decreasing with T.
  After each iteration, Adaboost decreases a particular upper-bound of the 0/1 training error. So in a long run, the training error will be pushed to zero.
- The classifier returned by AdaBoost is not guaranteed to have a good performance on the test set. In fact AdaBoost is prone to overfitting, unless it stops early.

Boosting: Forward Stagewise Additive Modeling

Consider an **Additive Model**:

$$f(x) = \sum_{t=1}^{T} \beta_t b(x; \gamma_t),$$

where $b(x; \gamma)$ is a classifier or a regression function characterized by parameter γ.

Forward Stagewise Optimization

(1) $f_0(x) = 0$

(2) For $t = 1$ to T,

- Given f_{t-1}, choose (β_t, γ_t) to minimize

$$\sum_i L\Big(y_i, f_{t-1}(x_i)\Big) + \beta b(x_i; \gamma); \qquad (1)$$

- Update $f_t(x) = f_{t-1}(x) + \beta_t b(x; \gamma_t)$.

Boosting algorithms can take various forms, depending on the choice of the base model $b(\cdot; \gamma)$, the choice of the loss function $L(y, f(x))$, and how optimization is done at (**??**).

AdaBoost is equivalent to forward stagewise additive modeling using an **exponential loss**

$$L(y, f(x)) = \exp(-yf(x)).$$

$$\begin{aligned} & \arg\min_{\beta,\gamma} \sum_i L(y_i, f_{t-1}(x_i) + \beta b(x_i;\gamma)) \\ = \quad & \arg\min_{\beta,\gamma} \sum_i \exp[-y_i f_{t-1}(x_i) - y_i \beta b(x_i;\gamma)] \\ = \quad & \arg\min_{\beta,\gamma} \sum_i w_i^{(t)} \exp(-\beta y_i b(x_i;\gamma)). \end{aligned}$$

- Instead of optimizing over both β and $b(\cdot, \gamma)$, AdaBoost just randomly picks a classifier $b(\cdot;\gamma)$, and then optimize over β.
- For any given $b(\cdot;\gamma)$, denote the corresponding weighted empirical error rate by ϵ, then the optimal β is given by

$$\beta = \frac{1}{2} \log \frac{1-\epsilon}{\epsilon}.$$

For regression, we can use L_2-**Boosting**.

- Loss function is the squared error,

$$\begin{aligned} & (y_i - f_{t-1}(x_i) - \beta b(x_i;\gamma))^2 \\ = \quad & (r_{it} - \beta b(x_i;\gamma))^2. \end{aligned}$$

- At the t-th iteration,

$$f_t(x) = f_{t-1}(x) + \hat{\beta}_t x^{(t)},$$

where $x^{(t)}$ denotes the variable (possibly random) chosen at the t-th iteration, and $\hat{\beta}_t$ is the estimated coefficient based on the partial residuals r_{it}.

When doing the optimization at the t-th iteration,

- for exponential loss, the effect of the previous $(t-1)$ functions becomes **weights**;
- for squared loss, the effect of the previous $(t-1)$ functions becomes **partial residuals**.

For many other loss functions, we don't have such a simple form for the effect of the previous $(t-1)$ functions, then we can approximate $L(y_i, f_{t-1}(x_i) + f_t(x_i))$ by Taylor expansions (**Gradient Boosting**).