Lecture 8

IDS575: Statistical Models and Methods
Theja Tulabandhula

Notes derived from the book titled "Elements of Statistical Learning [2nd edition] (Sections 9.2, Chapter 10)

Continuing from before, we will look at classification and regression methods that work with different (sometimes lesser) assumptions than linearity, but still give great predictive performance:

1. *(previously)* Generalized Additive Models (GAM),
2. *(today)* Tree-based methods,
3. *(today)* Adaboost and Gradient Boosting Methods,
4. *(future)* Random Forests,
5. *(future)* Multivariate Adaptive Regression Splines (MARS), and
6. *(future)* Support Vector Machines.

1 Tree-based Methods

These are methods that partition the input space into a set of rectangles and for each rectangle, make a constant prediction.

We will discuss a popular technique known as *CART (Classification and Regression Tree)* that can be used for regression and classification.

Example 1. An illustration of how a tree model looks like is given in Figure 1 for a 2-dimensional regression dataset.

Here, we are recursively doing a binary partitioning of the input space. First, we break it into two parts and model the response by the mean of y_i in each region. The choice of the

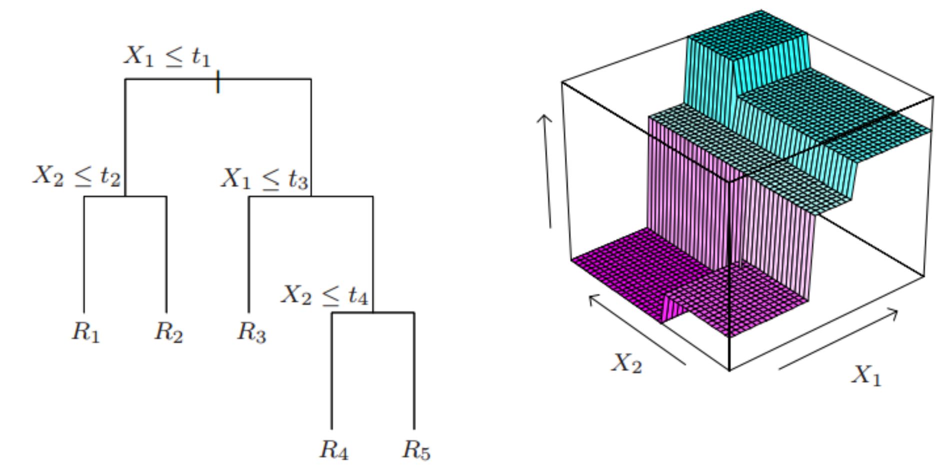


Figure 1: Partitioning of the input space in general (top left), partitioning using CART (top right), equivalent tree representation (bottom left), and predictions in each region (bottom right).

variable to split and where to split is based on some criteria (we will describe soon). Then we repeat the partitioning in these regions as long as a stopping condition is not met. In the figure, we get five regions. Our prediction function is:

$$\widehat{\mathrm{f}}(x) = \sum_{m=1}^{5} c_m 1[x \in R_m],$$

where c_ms are the constant predictions in each region.

One of the advantages of recursive binary tree is *interpretability*, even when we have $p > 2$ dimensions.

1.1 Regression Trees

Computing the best binary partition that minimizes the residual sum of squares (for example) is hard. So there is a greedy approach:

- Start with all data. Consider a split variable j and split point s. This defines regions $R_1(j,s) = \{X : X_j \leq s\}$ and $R_2(j,s) = \{X : X_j > s\}$. We can optimize for j, s by solving the following problem:

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right].$$

- For a choice of (j,s), the optimal $\hat{c}_k = \text{avg}(y_i | x_i \in R_k(j,s))$ for $k = 1,2$. Optimizing over this choice is not very difficult.
- We repeat the splitting for each of the regions obtained above.
- We stop the process when some criteria is met (see below).

Note 1. There are a couple of ways to stop the tree expansion process:

- We can stop if the decrease in the sum of squared errors is small after a split compared to before. This doesn't seem to work well in practice.
- We can stop if the number of observations in the current node is less than a number (say 5). Such a tree T_0 may be too big. This can be pruned using a strategy that is called *cost-complexity pruning.*
  - Notation: Let $T \subset T_0$ be a pruning of T_0 by collapsing any number of its internal nodes. Let the number of terminal nodes of any tree T be $|T|$. Let $N_m = |\{x_i \in R_m\}|$, $\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i$ and *node-impurity function* $Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$.
  - Define the cost-complexity function as $C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|$.
  - We can minimize this[1] for different α values that trade off tree size and its fit to training data (α itself can be chosen via K-fold cross-validation).

Note 2. The size of the tree is a design choice that controls predictive performance. A very large tree will potentially overfit the training data. Whereas a small tree may underfit.

[1] One can successively collapse the internal node that produces the smallest increase in $\sum_m N_m Q_m(T)$.

1.2 Classification Trees

When we have qualitative variable G, then we just need to replace squared loss and $Q_m(T)$ functions. Lets define the proportion of class k observations in a region R_m (node m) as:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} 1[y_i = k].$$

With this, a new observation in node m can be classified as $k(m) = \arg\max_k \hat{p}_{mk}$. The choice for $Q_m(T)$ can be any of the following:

- Mis-classification: $1 - \hat{p}_{mk(m)}$.
- Gini index: $\sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$.
- Cross-entropy: $-\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk})$.

Example 2. Here is a visualization in Figure 2 of how these look like when $K = 2$ and $(p_{m1}, p_{m2}) = (1 - p, p)$. if you can find a split such that p is closet to 0 or 1, the lower are these node-impurity functions.

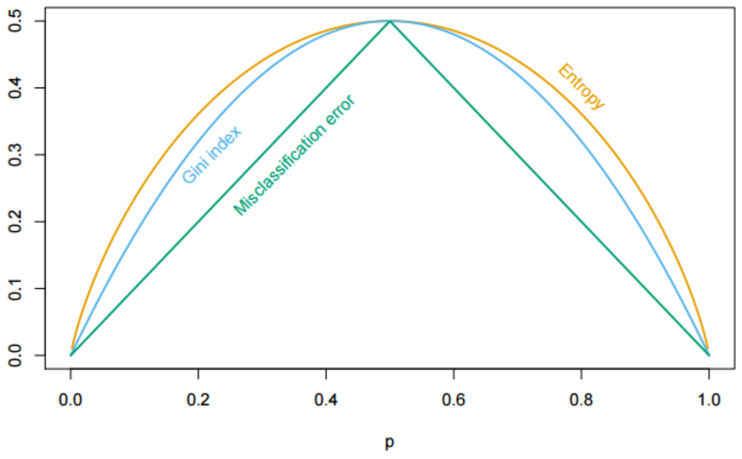


Figure 2: Node-impurity functions are higher when a split leads to higher error (p close to 0.5).

Note 3. There are many variants of tree-based methods such as C4.5, CART etc and we have only looked at one of them, namely, CART.

Example 3. Consider the spam classification problem. The performance of tree-based classifiers is shown in Figure 3. The pruned tree is shown in Figure 4.

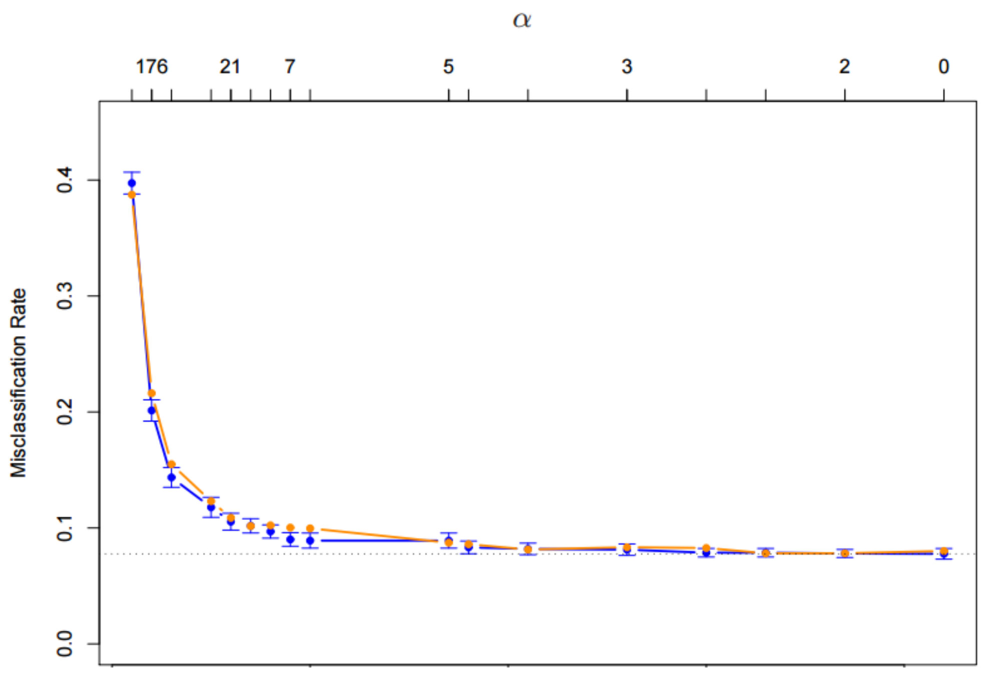


Figure 3: Spam classification: 10-fold CV performance is plotted (blue curve) as a function of α (top label). The orange curve is the test error.

1.3 Issues with Trees

There are several issues to be aware of when using tree-based methods for regression or classification tasks. These are:

- *Categorical predictors*: When X_j takes q values, then there are $2^{q-1} - 1$ partitions into two groups, which is a lot of choices. The impact of this is that the tree building process will be splitting on these choices more than the other variables. There is also a possibility of overfitting because of so many choices.

- *Binary splitting vs continuous splitting*: Although we could do multi-way splits, it tends to break data too quickly leading to worse predictive performance. One could also think of a split such as $\sum_j a_j X_j \geq s$, but these become non-interpretable very quickly.

- *Stability*: Often a small change in data will change the whole tree, especially if there is a change at the beginning stages of tree-building process.

- *Smoothness*: When the underlying regression function $E[Y|X = x]$ is smooth, then region-wise constant function fitting may not be a good idea.

Finally, lets also look at the issue of missing data deeply below, as it not only affects tree-based methods, but many other methods for supervised learning.

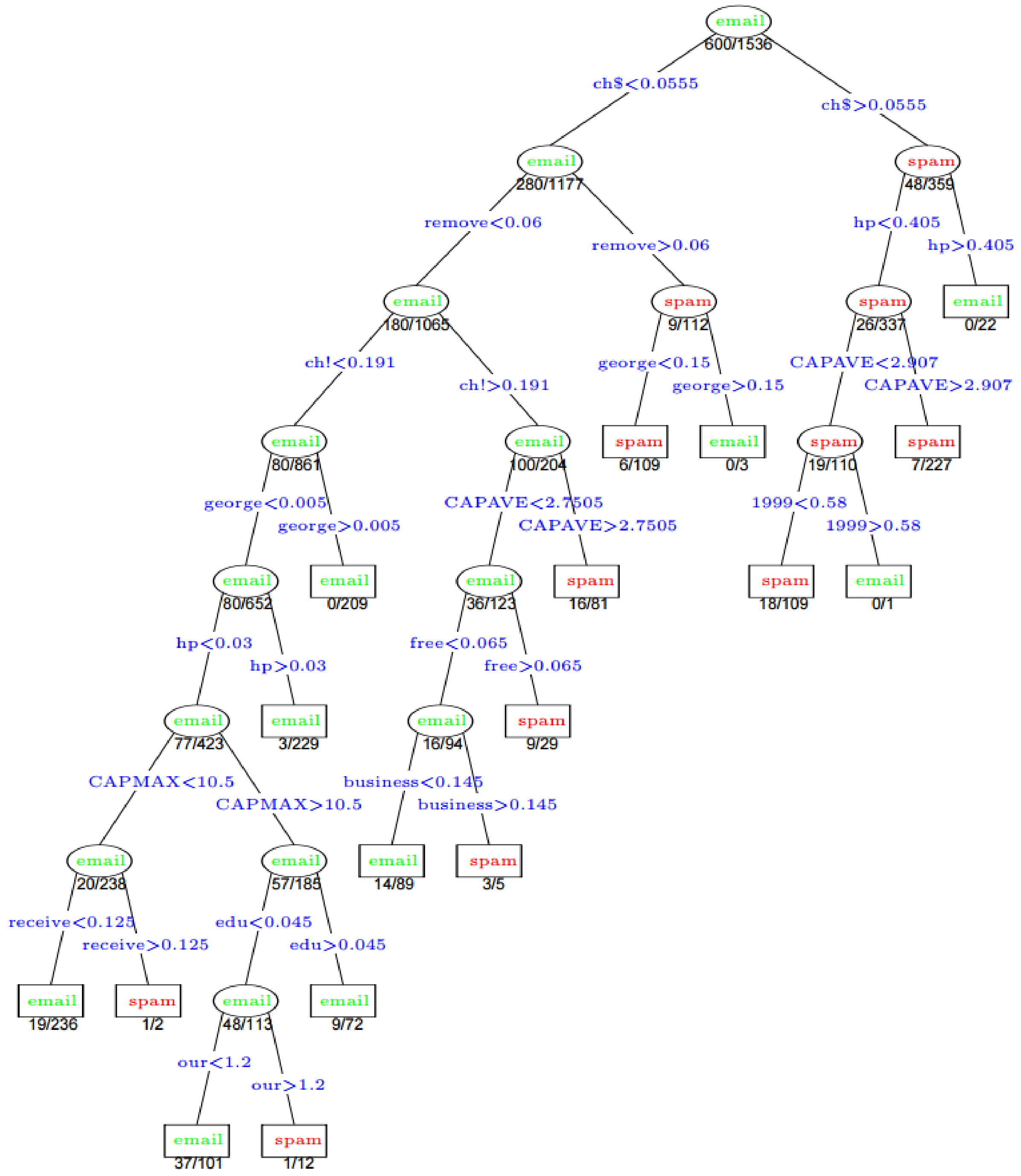


Figure 4: Spam classification: the pruned tree. The majority class is shown in each node. The ratio under the nodes indicate misclassification rates on test data.

1.4 Missing Data

Most importantly, we need to hypothesize whether the missing data influences the observed data in any way before choosing a solution approach.

Example 4. As a toy example, consider that physical measurements were made of Olympic athletes and the latter information was not recorded. Then any analysis based on this sample will be subject to selection bias.

Let $\mathbf{X}_{\text{obs}}$ be the observed entries in $\mathbf{X}$ and let $Z = (\mathbf{y}, \mathbf{X}), Z_{\text{obs}} = (\mathbf{y}, \mathbf{X}_{\text{obs}})$. Let $\mathbf{R}$ be an indicator matrix with ij^{th} entry 1 if x_{ij} is missing.

Data is *missing at random* (MAR) if $Pr(\mathbf{R}\,|\,\mathbf{Z}, \theta) = Pr(\mathbf{R}\,|Z_{\text{obs}}, \theta)$, where θ is a parameter for the distribution of $\mathbf{R}$.

Data is *missing completely at random* (MCAR) if $Pr(\mathbf{R}\,|\,\mathbf{Z}, \theta) = Pr(\mathbf{R}\,|\theta)$, where θ is again a parameter for the distribution of $\mathbf{R}$.

Example 5. If a patient's sickness measurement was not taken because she was too sick, then that observation would not be MAR or MCAR.

Here are some workarounds if data is MCAR:

- If some of the predictors have missing entries, then we can discard the corresponding observations if they are relatively small in number.
- Another option is to *impute*: fill in some reasonable values at these locations in the dataset. This is a popular approach in many settings.
- Depend on an algorithm such as EM to deal with missing values in the training phase.

Note 4. For tree-based methods, if the predictor with missing entries X_j is categorical, then we can just make a new category called 'missing' and proceed. Another way to deal with missing entries is to not consider them in the binary partitioning step. Further, during tree construction, we can track surrogate variable and split choices that mimic the original splits. This helps when during prediction, the new feature vector also has missing entries, then the surrogate variables and splits can be used.

2 Adaboost and Gradient Boosting

This is a method that combines the outputs of many *weak classifiers* in a way that has very good predictive performance, both for regression and classification. A weak classifier is one which is only slightly better than random guessing.

Lets directly jump into a boosting algorithm called *Adaboost.M1*. Let there be 2 classes such that $Y \in \{-1, 1\}$. We will use $G(X)$ to represent a classifier[2].

Intuitively boosting in this setting applies the weak classification algorithm on modified copies of the original training data producing classifiers $G_m(x)$ for $m = 1, ..., M$. The final classifier is $G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$. See Figure 5 for a schematic and Figure 6 for the algorithm.

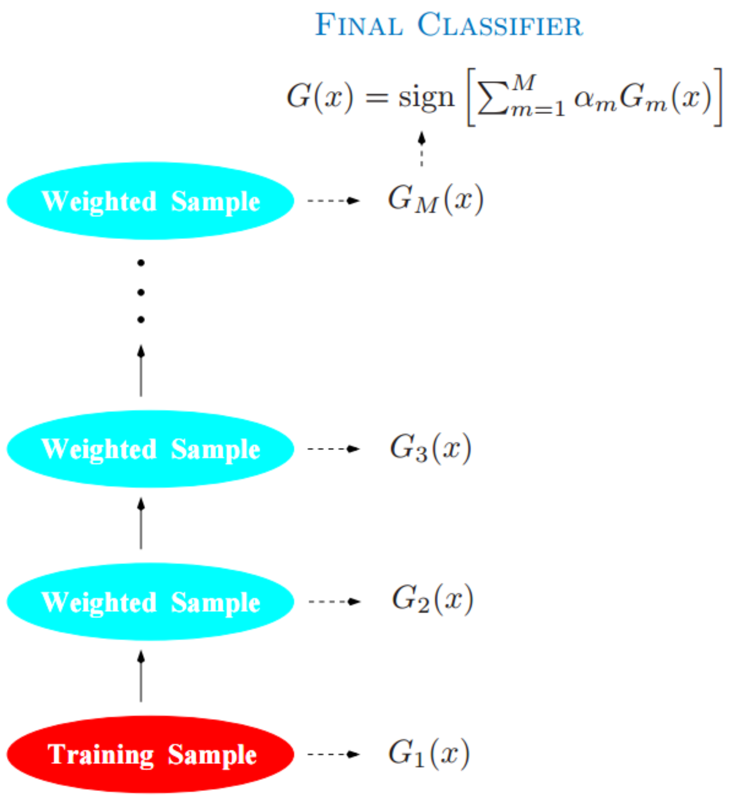
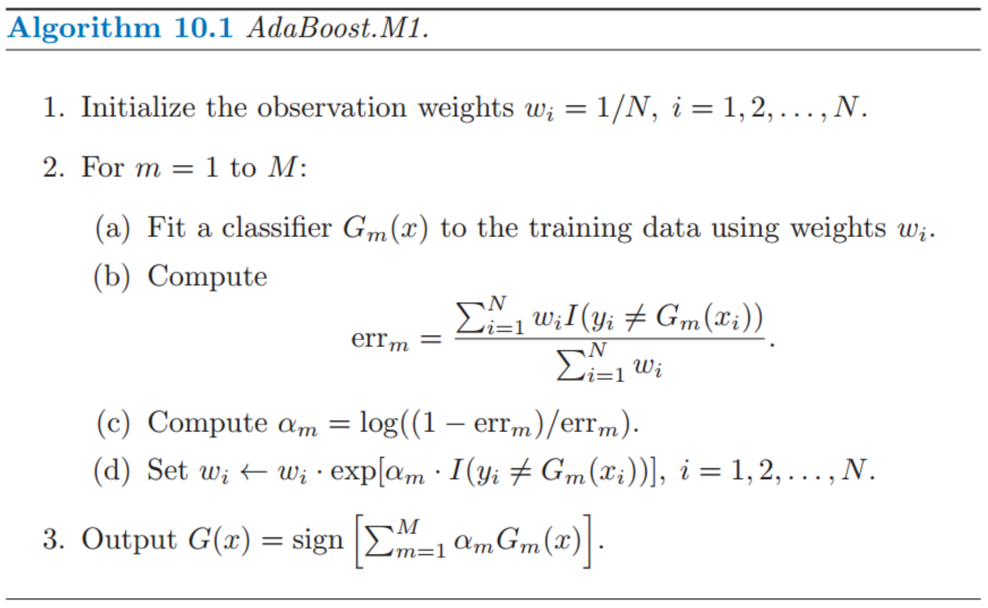


Figure 5: Adaboost.

[2]Notation: we had used G to represent qualitative output variable before, here we are using it as a classification function.



Algorithm 10.1 *AdaBoost.M1.*

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \dots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
            $\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$.
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \dots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.
```

Figure 6: Adaboost.M1 algorithm.

Example 6. Consider a synthetic data setup: $Y = 1$ if $\sum_{j=1}^{10} X_j^2 > 9.3$ and 0 otherwise. The X_js are independent standard Gaussians. Let $N = 2000$. The weak classifier is a two-node classification tree (also called a *stump*). As shown in Figure 7, the performance of the stump is 45.8%, and a 244-node tree is 24.7%, whereas boosting reaches 5.8% in 400 iterations.

2.1 Why does Boosting Work?

Boosting works because it fits an additive model with exponential loss.

In particular, it sequentially adds a new basis function (G_m) without adjusting the parameters and coefficients of those that have already been added (α_j, G_j for $j = 1, .., m-1$).

A general version of this stagewise additive model fitting is shown in Figure 8.

For Adaboost.M1, $L(y, f(x)) = \exp(-yf(x))$. Hence, we solve

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{N} \exp(-y_i f_{m-1}(x_i)) \exp(-\beta y_i G(x_i)).$$

From manipulating this expression a bit (we will not do that here), Adaboost.M1 can be seen as equivalent to forward stagewise additive modeling, which we know are good predictive models. A slight difference is that Adaboost is not minimizing training-set misclassification error (see Step 2(a) in figure 6 where no loss criteria is specified).

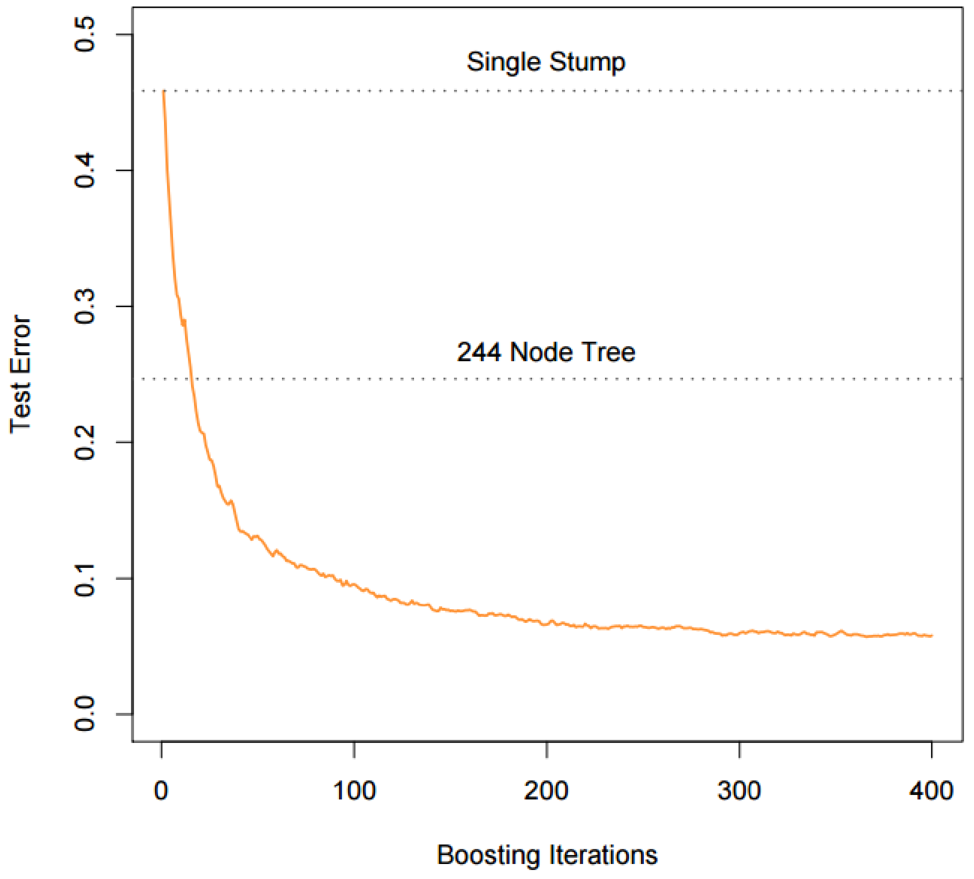


Figure 7: Boosting on toy data.

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

```
1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to $M$:
    (a) Compute
        $$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$
    (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
```

Figure 8: General stagewise fitting. Here $b(x_i; \gamma)$ is a basis function.

Example 7. A boosted tree is a sum of scaled trees:

$$f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m).$$

We would like to solve the following step in the forward stagewise procedure:

$$\Theta_m = \arg\min_{\Theta_m} \sum_{i=1}^{M} L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)),$$

where $\Theta_m = \{R_{jm}, \gamma_{jm}\}_1^{J_m}$ corresponds to the regions and the constants of the next tree given the current model $f_{m-1}(x)$. Finding the regions is difficult, while finding constants is relatively straightforward:

$$\hat{\gamma}_{jm} = \arg\min_{\gamma_{jm}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma_{jm}). \qquad (1)$$

We get Adaboost with trees (Figure 6) when the problem is a 2-class classification problem with exponential loss function.

Some advantages of tree methods are sacrificed by boosting: computational speed, interpretability and robustness against mislabeling of training data. Gradient boosted (tree) models attempt to mitigate these issues.

In fact, we will focus on speed below. When the loss function is general (not just least squares or exponential), performing stagewise boosting is computationally difficult, so we use gradient boosting as a way to get around this.

2.2 Gradient Boosted Models (GBMs)

Lets now motivate gradient boosting by thinking about gradients in a generic problem. Say $L(f_m) = \sum_{i=1}^{N} L(y_i, f_m(x_i))$. Instead of thinking of function f_m, lets think of vector $\mathbf{f}_m \in \mathbb{R}^N$, where $\mathbf{f}_m = [f_m(x_1), ..., f_m(x_N)]^T$.

We can think of these as N numbers that need to be set to minimize $L(f_m)$. And there is a family of methods (actually gradient based methods) that do this by iteratively updating $\mathbf{f}_m$ from $\mathbf{f}_{m-1}$ using gradient information.

For example, there is a numerical method called steepest descent which does the following:

- Computes $\mathbf{g}_m \in \mathbb{R}^N$ such that

$$g_{im} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x_i) = f_{m-1}(x_i)}.$$

- Computes *step length* ρ_m such that

$$\rho_m = \arg\min_{\rho} L(\mathbf{f}_{m-1} - \rho\, \mathbf{g}_m).$$

- Then updates $\mathbf{f}_m = \mathbf{f}_{m-1} - \rho_m\, \mathbf{g}_m$ and repeats.

The intuition behind steepest descent is that $-\mathbf{g}_m$ is the local direction in $\mathbb{R}^N$ that decreases the objective $L(f_m)$ from the current point $\mathbf{f}_{m-1}$.

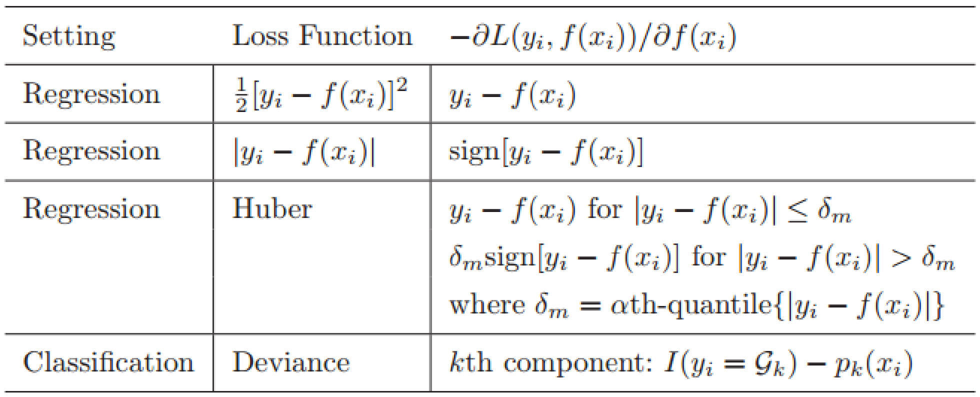
Of course, these N numbers in vector $\mathbf{f}_m$ are dependent on each other through a model, such as a tree. This brings us to the next idea: building a tree (or any other model) that approximates these numbers ($\mathbf{f}_m$).

Tree predictions $T(x_i;\Theta_m)$ are analogous to components of the negative gradient. And thus the N numbers are constrained to be predictions of a tree.

If we find such a tree, then finding the γ_{jm} is similar to finding ρ_m above. In the former, we can think of doing J_m line searches, one for each region.

So the key idea is to induce a tree $T(x;\Theta_m)$ at the m^{th} iteration whose predictions are close to the negative gradient , say by using a least squares fit. Once the tree is fit, the constants in each region are fit using Equation 1.

Gradients for commonly used loss functions are given in Figure 9.

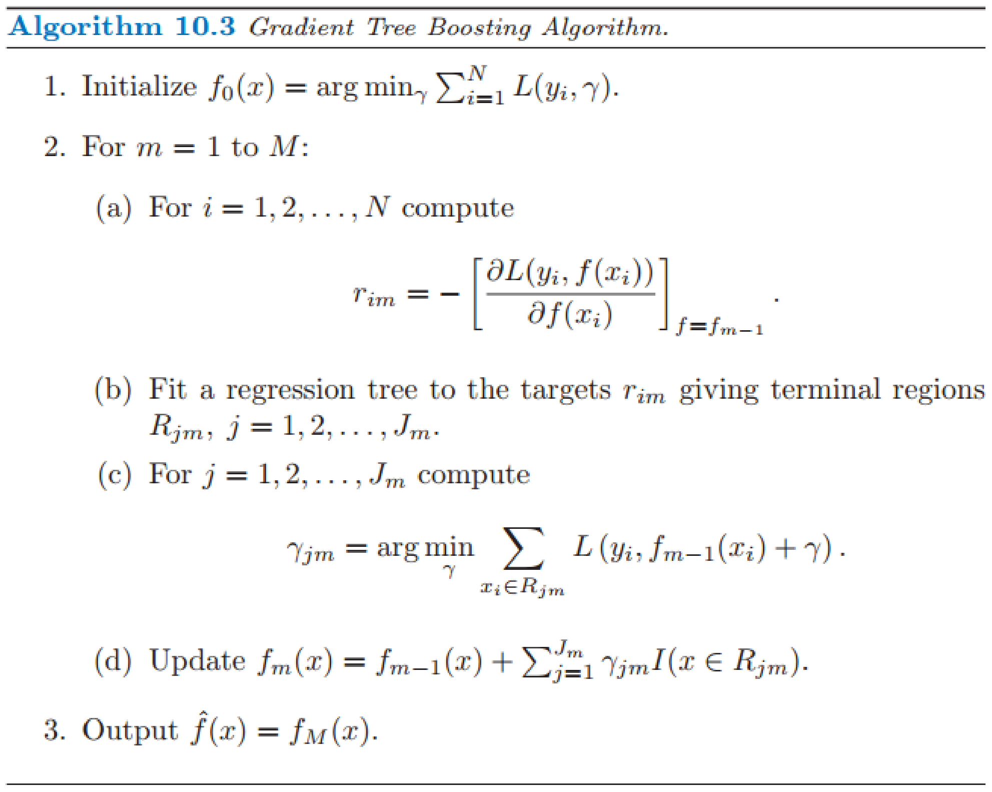


Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \le \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha$th-quantile$\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Figure 9: Gradients.

Figure 10 presents the gradient tree-boosting algorithm for regression, which summarizes what we discussed above.

Note 5. The key parameters for this method are: (a) the number of iterations M, and (b) the size of trees $J_m, m = 1, .., M$.



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 10: Gradient boosting algorithm for trees.

2.3 Gradient Boosting for Classification

For classification, K trees are fitted at each iteration. Tree T_{km} is fit to its negative gradient vector $\mathbf{g}_{km}$ which component wise is given as:

$$-g_{ikm} = \left[\frac{\partial L(y_i, f_1(x_i), \ldots, f_K(x_i))}{\partial f_k(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}.$$

3 Summary

We learned the following things:

- Tree-based methods.
- Idea of boosting to get better predictive performance.

A Sample Exam Questions

1. What is Gini index in the context of a classification-tree?

2. What are the different models of missing data?

3. Describe the Gradient Boosted Tree algorithm. What is the use of gradients?

B Loss Functions

In the classification setting, loss function L should ideally penalize incorrect classifications. A generalized notion of this is to penalize negative *margins* (margin is defined as $yf(x)$) instead of misclassifications. See Figure 11 for some examples.

Example 8. Misclassification loss $L(y, f(x)) = 1[yf(x) < 0]$ where $1[\cdot]$ is an indicator function (takes value 1 when the argument is true).

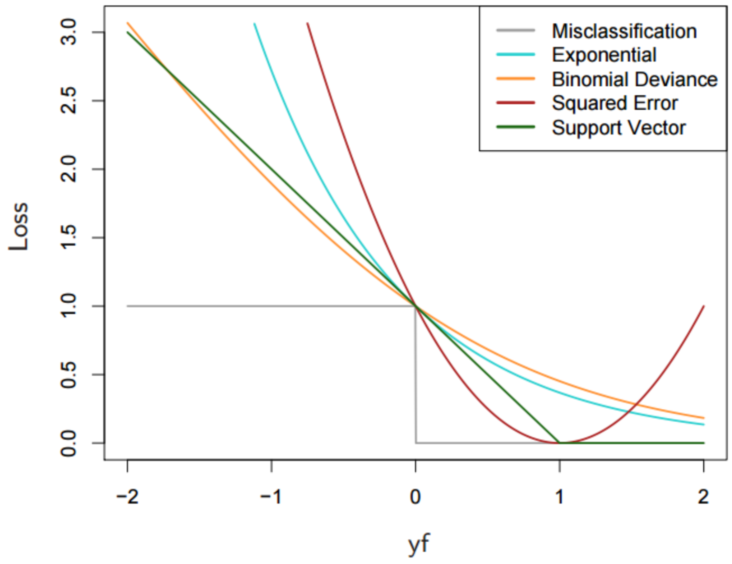


Figure 11: Loss functions for classification vs margin.

A similar set of loss functions for regression are shown in Figure 12.

C Prediction Models

Here is a table (Figure 13) that shows the relative merits of various prediction models.

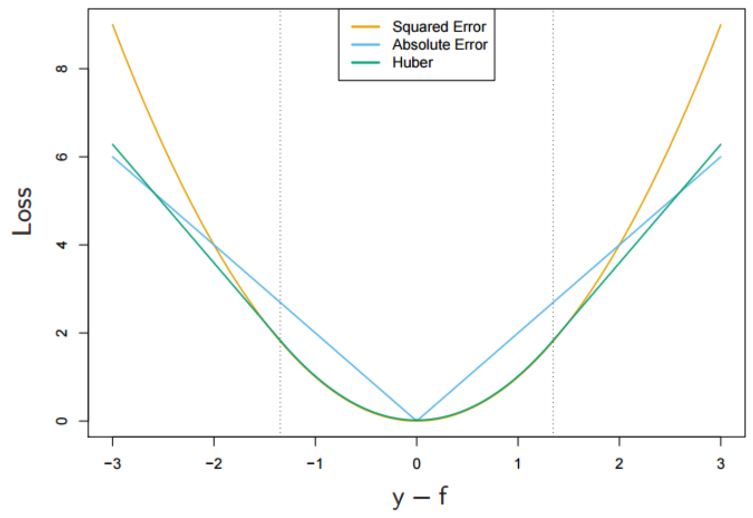


Figure 12: Loss functions for regression vs $y - f(x)$.

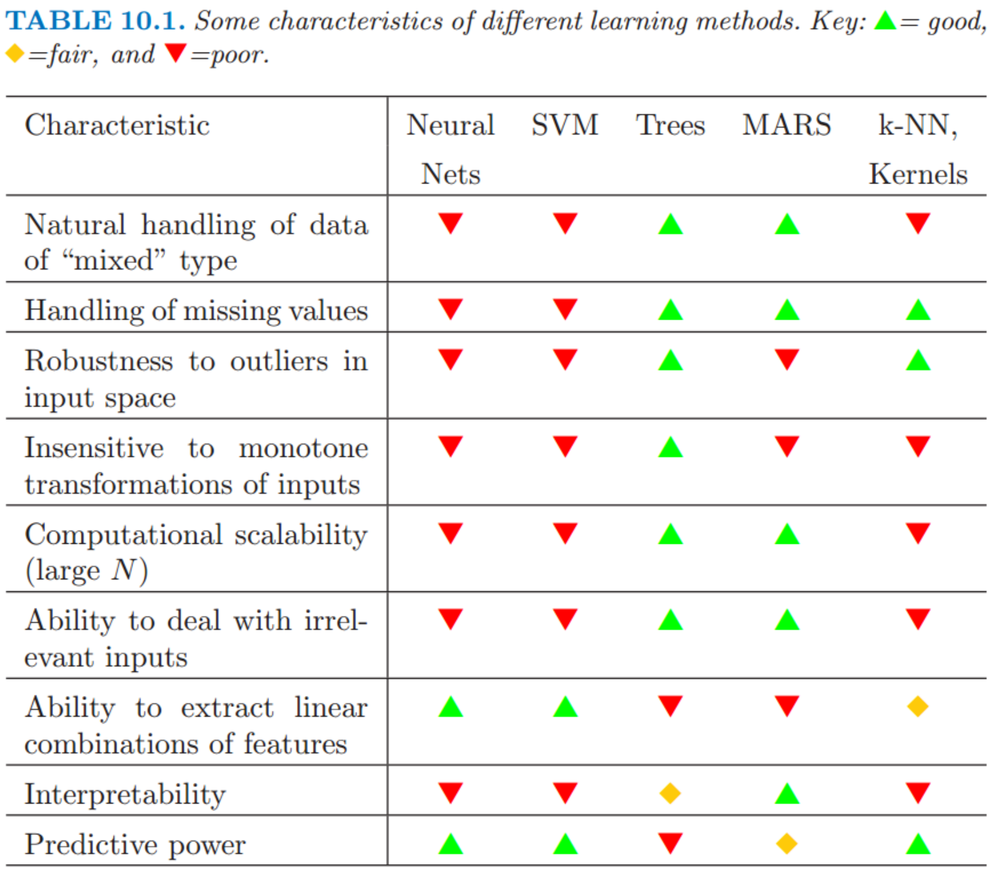


TABLE 10.1. *Some characteristics of different learning methods. Key:* ▲*= good,* ◆*=fair, and* ▼*=poor.*

Characteristic	Neural Nets	SVM	Trees	MARS	k-NN, Kernels
Natural handling of data of "mixed" type	▼	▼	▲	▲	▼
Handling of missing values	▼	▼	▲	▲	▲
Robustness to outliers in input space	▼	▼	▲	▼	▲
Insensitive to monotone transformations of inputs	▼	▼	▲	▼	▼
Computational scalability (large N)	▼	▼	▲	▲	▼
Ability to deal with irrelevant inputs	▼	▼	▲	▲	▼
Ability to extract linear combinations of features	▲	▲	▼	▼	◆
Interpretability	▼	▼	◆	▲	▼
Predictive power	▲	▲	▼	◆	▲

Figure 13: List of prediction models.