Chapter 3: Decision tree methods
Chapter 4: Boosting

Esa Ollila

Department of Signal Processing and Acoustics
Aalto University, Finland

Large Scale Data Analysis / Aalto University

Menu

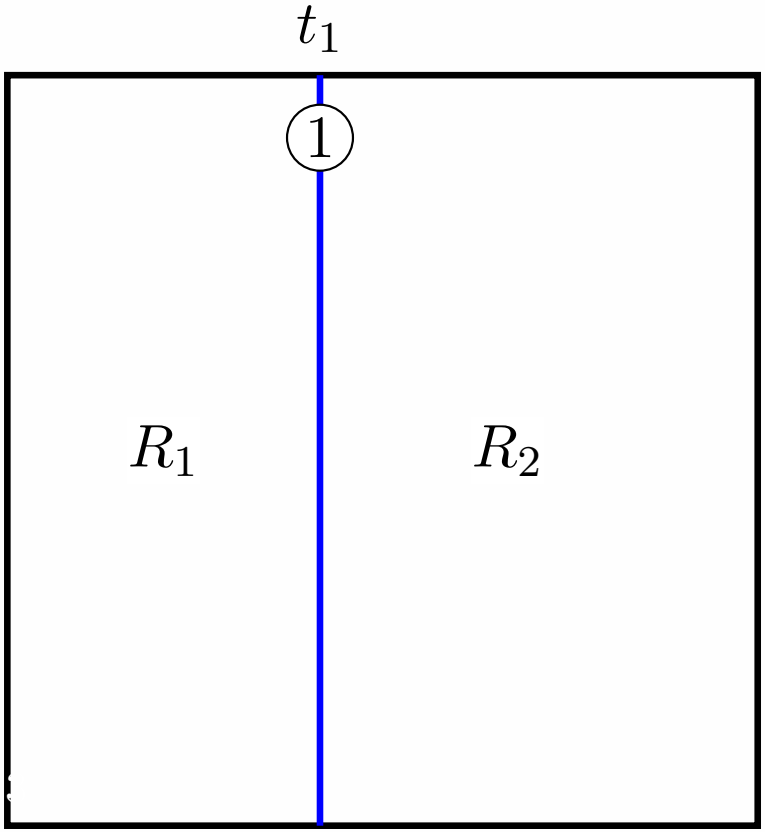
Recursive binary tree

- Decision trees can be applied to both regression and classification.
- For a moment, consider the regression task ($Y \in \mathbb{R}$)
- Recursive binary trees (single feature splits) are most commonly used:

1. Split the space into 2 regions.
   - Choose the variable and split-point achieving the best fit
   - "Fit" is based on training data and some loss fnc., e.g., $(y - f(\mathbf{x}))^2$.
2. Model the response by the mean of Y in each region.
3. One or both of these regions are split into two more regions.
4. and this process is continued, until some stopping rule is applied.

Recursive binary tree: an example

Suppose we have only two features so $X = (X_1, X_2)$, each taking values in the unit interval. Hence the feature space is a square.

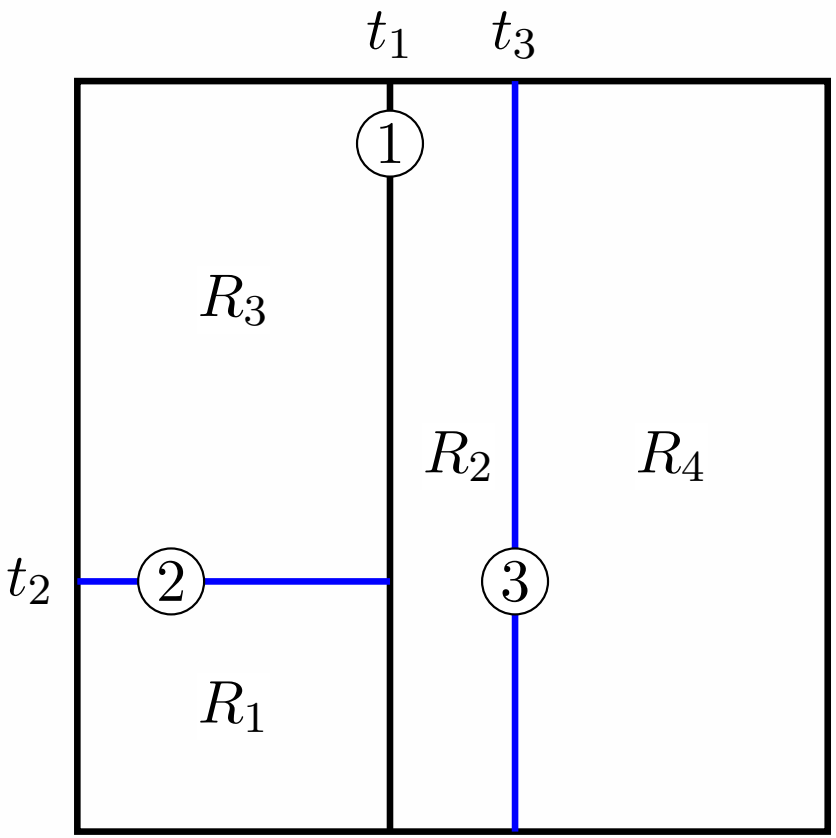


- Split at $X_1 = t_1$.
- Split the region $X_1 \leq t_1$ at $X_2 = t_2$ and $X_1 > t_1$ at $X_1 = t_3$.
- Split the region $X_2 > t_2$ and $X_1 \leq t_1$ at $X_1 = t_4$.
- Split the region $X_1 > t_3$ at $X_2 = t_5$.

$\Rightarrow$ partitions (X_1, X_2)-space into regions $R_1, \ldots, R_5, R_6$.

Recursive binary tree: an example

Suppose we have only two features so $X = (X_1, X_2)$, each taking values in the unit interval. Hence the feature space is a square.

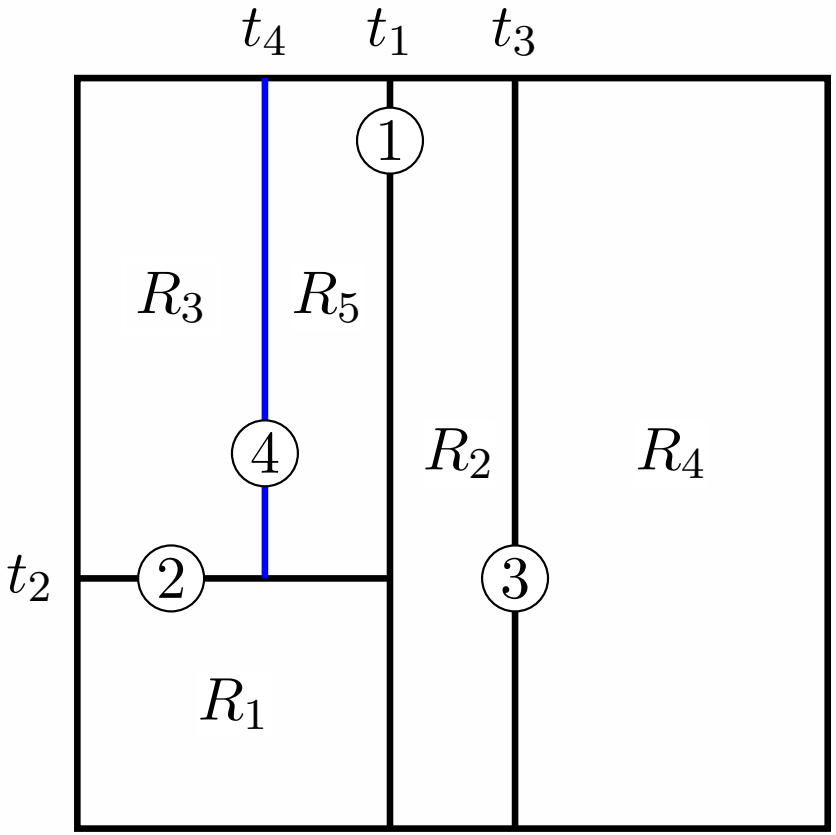


- Split at $X_1 = t_1$.
- Split the region $X_1 \leq t_1$ at $X_2 = t_2$ and $X_1 > t_1$ at $X_1 = t_3$.
- Split the region $X_2 > t_2$ and $X_1 \leq t_1$ at $X_1 = t_4$.
- Split the region $X_1 > t_3$ at $X_2 = t_5$.

$\Rightarrow$ partitions (X_1, X_2)-space into regions $R_1, \ldots, R_5, R_6$.

Recursive binary tree: an example

Suppose we have only two features so $X = (X_1, X_2)$, each taking values in the unit interval. Hence the feature space is a square.

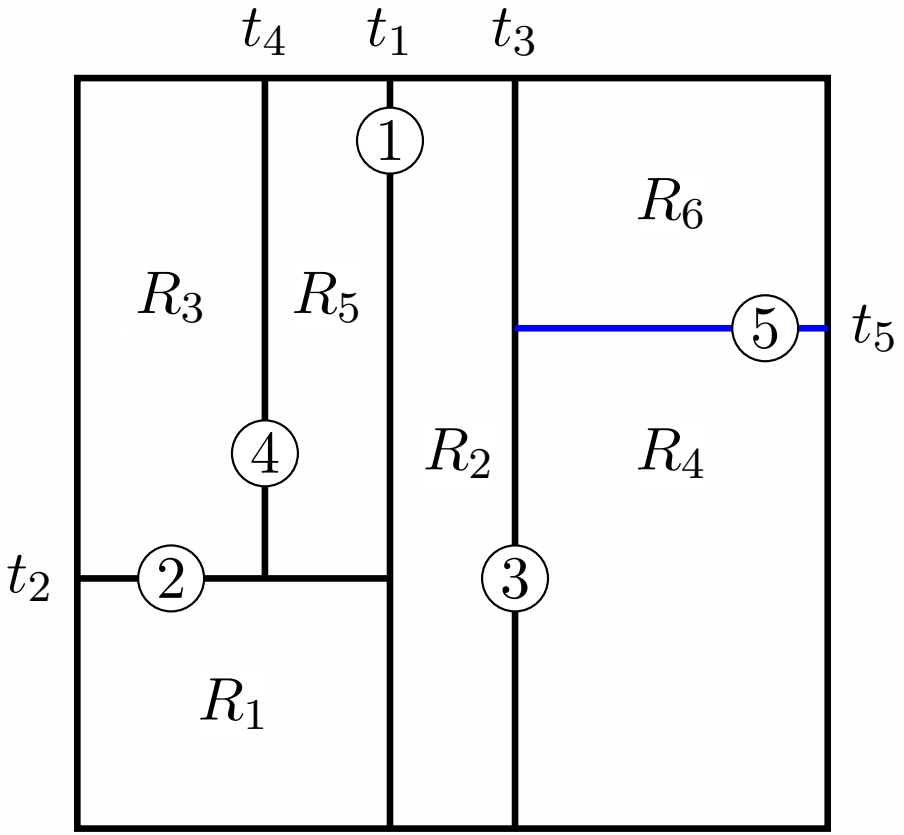


- Split at $X_1 = t_1$.
- Split the region $X_1 \leq t_1$ at $X_2 = t_2$ and $X_1 > t_1$ at $X_1 = t_3$.
- Split the region $X_2 > t_2$ and $X_1 \leq t_1$ at $X_1 = t_4$.
- Split the region $X_1 > t_3$ at $X_2 = t_5$.

$\Rightarrow$ partitions (X_1, X_2)-space into regions $R_1, \ldots, R_5, R_6$.

Recursive binary tree: an example

Suppose we have only two features so $X = (X_1, X_2)$, each taking values in the unit interval. Hence the feature space is a square.

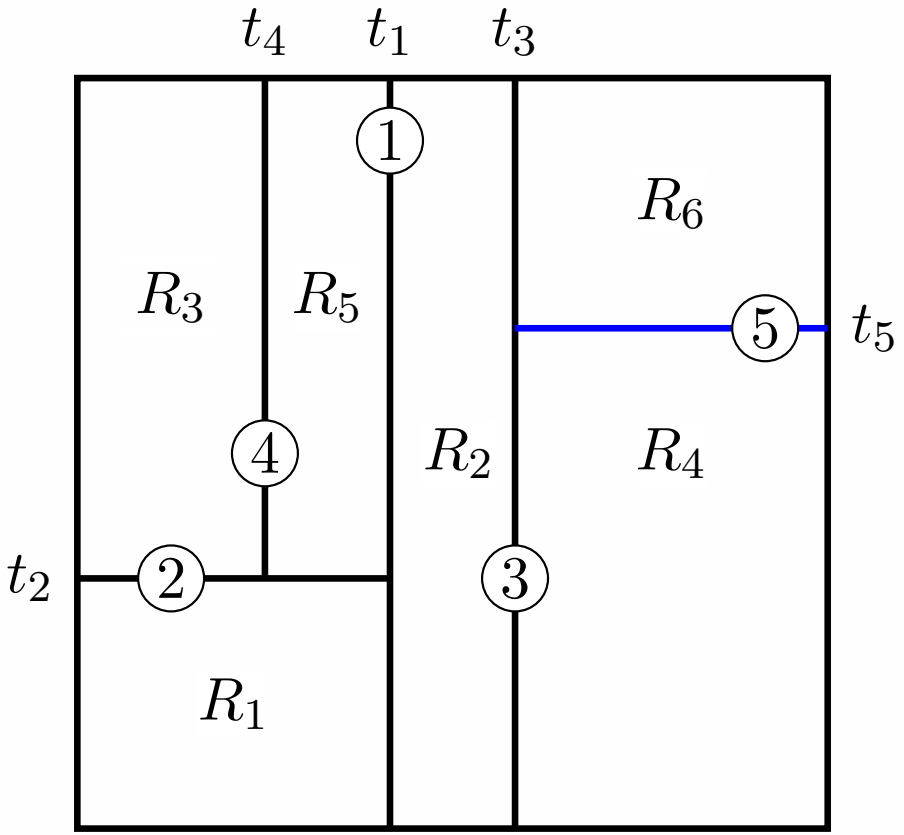


- Split at $X_1 = t_1$.
- Split the region $X_1 \leq t_1$ at $X_2 = t_2$ and $X_1 > t_1$ at $X_1 = t_3$.
- Split the region $X_2 > t_2$ and $X_1 \leq t_1$ at $X_1 = t_4$.
- Split the region $X_1 > t_3$ at $X_2 = t_5$.

$\Rightarrow$ partitions (X_1, X_2)-space into regions $R_1, \ldots, R_5, R_6$.

Recursive binary tree: an example

Suppose we have only two features so $X = (X_1, X_2)$, each taking values in the unit interval. Hence the feature space is a square.

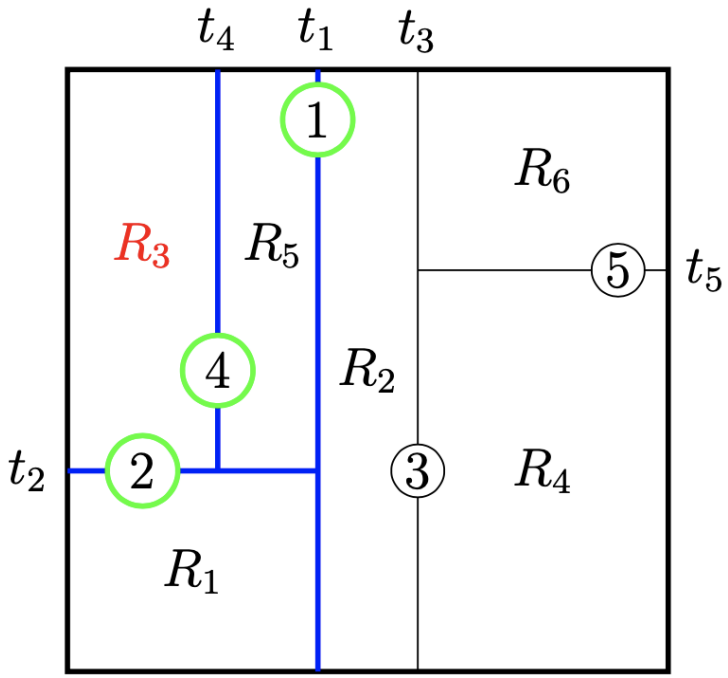


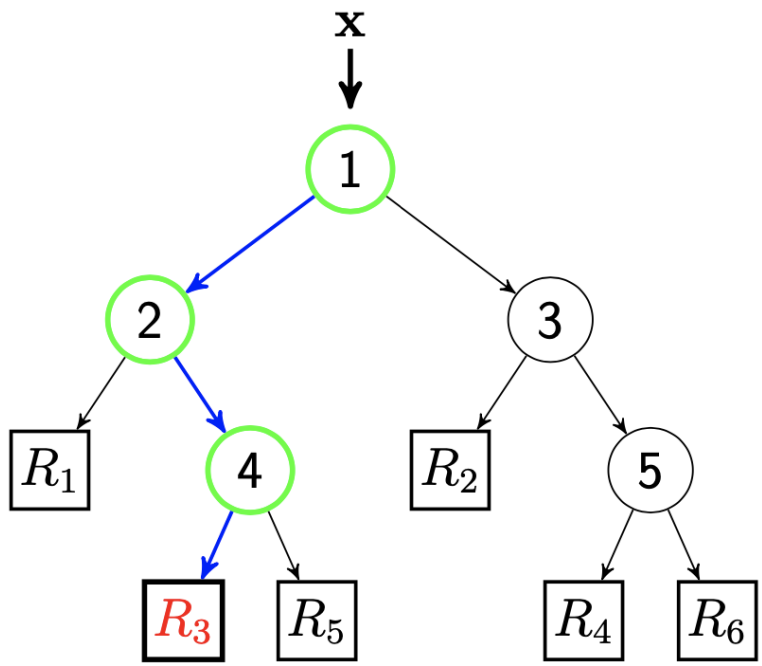
- Split at $X_1 = t_1$.
- Split the region $X_1 \leq t_1$ at $X_2 = t_2$ and $X_1 > t_1$ at $X_1 = t_3$.
- Split the region $X_2 > t_2$ and $X_1 \leq t_1$ at $X_1 = t_4$.
- Split the region $X_1 > t_3$ at $X_2 = t_5$.

$\Rightarrow$ partitions (X_1, X_2)-space into regions $R_1, \ldots, R_5, R_6$.

Recursive binary tree: an example

The same model can be represented by the binary tree (right panel):

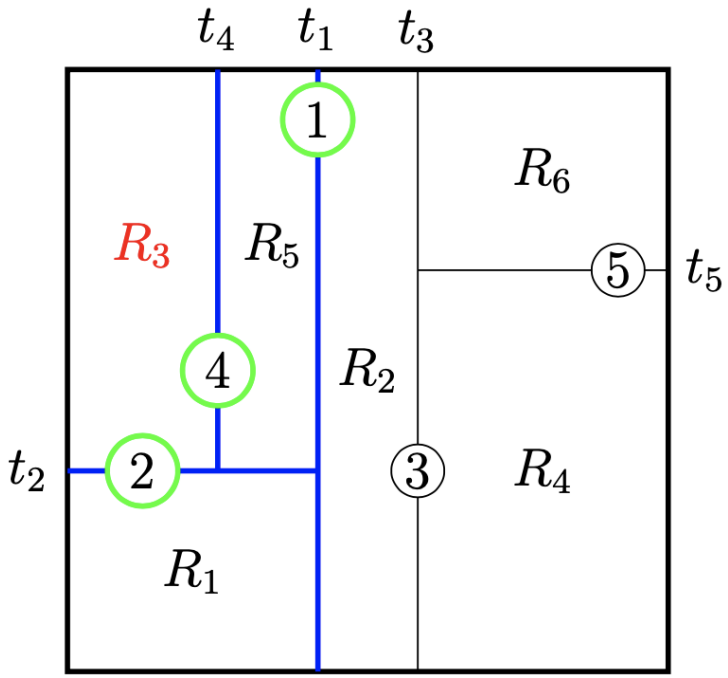


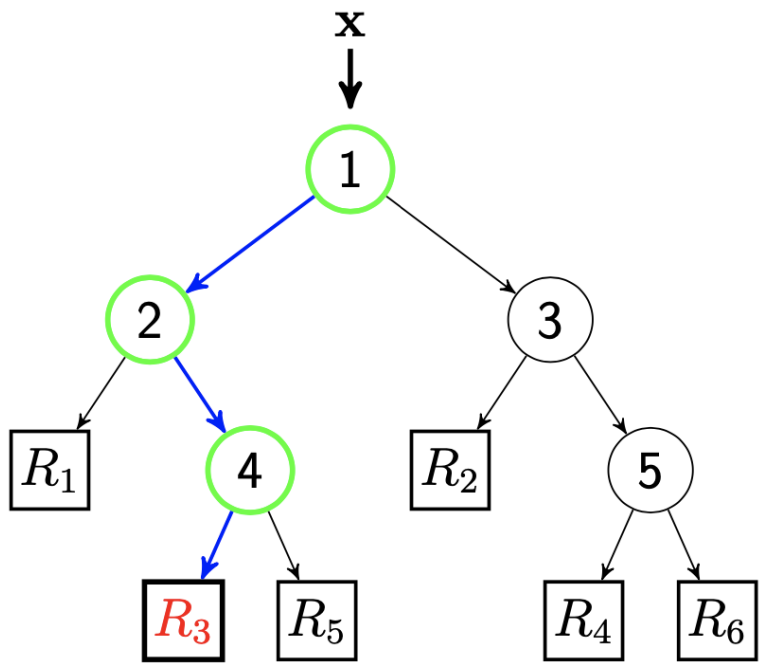


- We predict the response for a given test data $\mathbf{x}$ using the mean of the training data in the region to which it belongs.
- On the right panel, an observation $\mathbf{x}$ belong to region R_3, thus its prediction is $\hat{y} = \text{ave}(y_i \,|\, \mathbf{x}_i \in R_3)$.

Recursive binary tree: an example

The same model can be represented by the binary tree (right panel):





- We predict the response for a given test data $\mathbf{x}$ using the mean of the training data in the region to which it belongs.
- On the right panel, an observation $\mathbf{x}$ belong to region R_3, thus its prediction is $\hat{y} = \text{ave}(y_i \,|\, \mathbf{x}_i \in R_3)$.

What kinds of trees are there?

- There are two types of trees:
  1. **Regression trees**: continuous output $Y \in \mathbb{R}$.
     Matlab: `fitrtree` (statistics and machine learning toolbox).
     Python: `DecisionTreeRegressor` (scikit-learn: trees)
  2. **Classification trees**: categorical output $Y \in \mathbb{R}$.
     Matlab: `fitctree` (statistics and machine learning toolbox).
     Python: `DecisionTreeClassifier` (scikit-learn: trees)
- These differ mainly on criteria for splitting nodes and pruning the tree.

Tree terminology

Tree is denoted by T:

- Terminal nodes or leaves of the tree T correspond to the regions $R_1, R_2, \ldots, R_6$.
- The points along the tree where the predictor space is split are referred to as internal nodes.
- Tree size, denoted by $|T|$ or M, is the number of terminal nodes (previous example $M = 6$).
- Subtree $T \subset T_0$ to be any tree that can be obtained by pruning T_0, that is, collapsing any number of its internal (non-terminal) nodes.
- Stump is a tree with only two terminal nodes ($M = 2$).

Menu

Regression trees

- Given the data of inputs and and a response,

$$\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \qquad \text{with } \mathbf{x}_i^\top = (x_{i1}, \ldots, x_{ip}),$$

  the algorithm needs to decide:

  1. *What is the best split?* (which variables and split points)
  2. *How to split the variables*, i.e., what topology (shape) the tree should have? E.g., binary tree or multiway?
  3. *How large to grow the tree?*

- Overfitting is easy:
  - if we grow a tree deep enough, we can fit the training data perfectly.

Regression trees

- Given the data of inputs and and a response,

$$\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \qquad \text{with } \mathbf{x}_i^\top = (x_{i1}, \ldots, x_{ip}),$$

  the algorithm needs to decide:

  1. *What is the best split?* (which variables and split points)
  2. *How to split the variables*, i.e., what topology (shape) the tree should have? E.g., binary tree or multiway?
  3. *How large to grow the tree?*

- Overfitting is easy:
  - if we grow a tree deep enough, we can fit the training data perfectly.

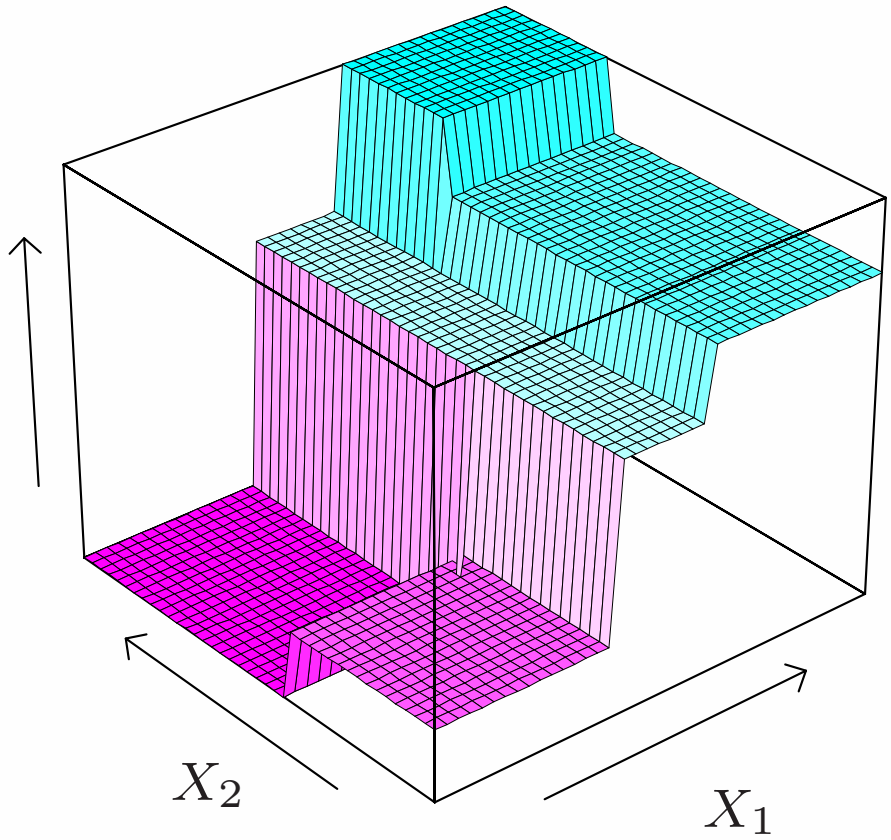
Regression trees (cont'd)

- Suppose we have a partition $R_1, R_2, \ldots, R_M$.
- Then we model the response as a constant c_m in each region:

$$f(\mathbf{x}) = c_m \sum_{m=1}^{M} 1_{\{\mathbf{x} \in R_m\}}.$$

In the binary regression tree, we may obtain the following prediction surface (here we have 5 leaves).

Figure from Hastie et al. [2009]



Regression trees (cont'd)

- If we we us the sum of squares $\sum_i (y_i - f(\mathbf{x}_i))^2$ as the criterion to minimize, the best $\hat{c}_m$ is

$$\hat{c}_m = \text{ave}(y_i | \mathbf{x}_i \in R_m) = \text{ average of } y_i \text{ in region } R_m.$$

- Finding the best binary partition in terms of minimum sum of squares is generally computationally infeasible.

$\rightarrow$ use a greedy algorithm explained in next slide

Greedy algorithm for splitting

1. Starting with all of the data, split variable j at a split point s, and define the pair of half-planes:

$$R_1(j,s) = \{\mathbf{x}|x_j \leq s\} \quad \text{and} \quad R_2(j,s) = \{\mathbf{x}|x_j > s\}$$

2. Determine the best splitting variable j and a split point s that solves

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

where for any j and s, the inner minimization is solved by

$$\hat{c}_1 = \text{ave}(y_i|\mathbf{x}_i \in R_1(j,s)) \quad \text{and} \quad \hat{c}_2 = \text{ave}(y_i|\mathbf{x}_i \in R_2(j,s))$$

3. Repeat the splitting process on each of the two regions. This process is continued until stopping criterion is met.

Greedy algorithm for splitting

1. Starting with all of the data, split variable j at a split point s, and define the pair of half-planes:

$$R_1(j,s) = \{\mathbf{x}|x_j \leq s\} \quad \text{and} \quad R_2(j,s) = \{\mathbf{x}|x_j > s\}$$

2. Determine the best splitting variable j and a split point s that solves

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

where for any j and s, the inner minimization is solved by

$$\hat{c}_1 = \text{ave}(y_i|\mathbf{x}_i \in R_1(j,s)) \quad \text{and} \quad \hat{c}_2 = \text{ave}(y_i|\mathbf{x}_i \in R_2(j,s))$$

3. Repeat the splitting process on each of the two regions. This process is continued until stopping criterion is met.

Greedy algorithm for splitting

1. Starting with all of the data, split variable j at a split point s, and define the pair of half-planes:

$$R_1(j,s) = \{\mathbf{x}|x_j \leq s\} \quad \text{and} \quad R_2(j,s) = \{\mathbf{x}|x_j > s\}$$

2. Determine the best splitting variable j and a split point s that solves

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

   where for any j and s, the inner minimization is solved by

$$\hat{c}_1 = \text{ave}(y_i|\mathbf{x}_i \in R_1(j,s)) \quad \text{and} \quad \hat{c}_2 = \text{ave}(y_i|\mathbf{x}_i \in R_2(j,s))$$

3. Repeat the splitting process on each of the two regions. This process is continued until stopping criterion is met.

Note: For each splitting variable, the determination of the split point s can be done very quickly and hence determination of the best pair (j,s) is feasible.

How large tree?

Q: What is a good stopping criterion for splitting:

- **how large we should grow the tree**?
- How do we know if we should split the tree node?

- If we grow a tree deep enough, we end up in massive overfitting.
- The tree size, M, i.e., the number of terminal nodes, is a tuning parameter governing the model's complexity, and the optimal tree size should be adaptively chosen from the data, e.g., via **pruning** .

Pruning the three

- We index terminal nodes by m (so m representing region R_m). and define

$$N_m = \#\{\mathbf{x}_i \in R_m\}, \qquad (= \text{node size})$$

$$\hat{c}_m = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} y_i,$$

$$Q_m(T) = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} (y_i - \hat{c}_m)^2.$$

- The cost complexity criterion is then defined as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

where $\alpha \geq 0$ is a tuning parameter that governs the tradeoff between tree size and its goodness of fit to the data:

- Large α results in smaller trees T, and conversely.

Pruning the three

- We index terminal nodes by m (so m representing region R_m). and define

$$N_m = \#\{\mathbf{x}_i \in R_m\}, \qquad (= \text{node size})$$

$$\hat{c}_m = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} y_i,$$

$$Q_m(T) = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} (y_i - \hat{c}_m)^2.$$

- The cost complexity criterion is then defined as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

where $\alpha \geq 0$ is a tuning parameter that governs the tradeoff between tree size and its goodness of fit to the data:

  - Large α results in smaller trees T (and vice versa).
  - When $\alpha = 0$ the solution is the full tree T_0.

Pruning the three

- We index terminal nodes by m (so m representing region R_m). and define

$$N_m = \#\{\mathbf{x}_i \in R_m\}, \qquad (= \text{node size})$$

$$\hat{c}_m = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} y_i,$$

$$Q_m(T) = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} (y_i - \hat{c}_m)^2.$$

- The cost complexity criterion is then defined as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

where $\alpha \geq 0$ is a tuning parameter that governs the tradeoff between tree size and its goodness of fit to the data:

  - Large α results in smaller trees T (and vice versa).
  - When $\alpha = 0$ the solution is the full tree T_0.

Pruning the three

- We index terminal nodes by m (so m representing region R_m). and define

$$N_m = \#\{\mathbf{x}_i \in R_m\}, \qquad (= \text{node size})$$

$$\hat{c}_m = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} y_i,$$

$$Q_m(T) = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} (y_i - \hat{c}_m)^2.$$

- The cost complexity criterion is then defined as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

where $\alpha \geq 0$ is a tuning parameter that governs the tradeoff between tree size and its goodness of fit to the data:

  - Large α results in smaller trees T (and vice versa).
  - When $\alpha = 0$ the solution is the full tree T_0.

Cost-complexity pruning

1. Grow a large tree T_0, stopping the splitting process only when some minimum node size (say 5) is reached.
2. For each α, find the subtree T_α that minimizes $C_\alpha(T)$:

$$T_\alpha = \arg\min_{T \subseteq T_0} \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

   using weakest-link pruning[1]:
   - 2a successively collapse the internal node that produces the smallest per-node increase in the fit $:= \sum_m N_m Q_m(T)$
   - 2b continue until one reaches the single-node (root) tree
   - 2c this produces a sequence of subtrees which contains T_α.
3. Estimation of α is achieved by five- or tenfold cross-validation: $\hat{\alpha}$ minimizes the cross-validated sum of squares.
4. Final tree is $T_{\hat{\alpha}}$.

Cost-complexity pruning

1. Grow a large tree T_0, stopping the splitting process only when some minimum node size (say 5) is reached.
2. For each α, find the subtree T_α that minimizes $C_\alpha(T)$:

$$T_\alpha = \arg\min_{T \subseteq T_0} \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

   using weakest-link pruning[1]:

   2a. successively collapse the internal node that produces the smallest per-node increase in the fit $:= \sum_m N_m Q_m(T)$
   2b. continue until one reaches the single-node (root) tree
   2c. this produces a sequence of subtrees which contains T_α.

3. Estimation of α is achieved by five- or tenfold cross-validation: $\hat{\alpha}$ minimizes the cross-validated sum of squares.
4. Final tree is $T_{\hat{\alpha}}$.

Cost-complexity pruning

1. Grow a large tree T_0, stopping the splitting process only when some minimum node size (say 5) is reached.
2. For each α, find the subtree T_α that minimizes $C_\alpha(T)$:

$$T_\alpha = \arg\min_{T \subseteq T_0} \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

   using weakest-link pruning[1]:

   2a. successively collapse the internal node that produces the smallest per-node increase in the fit $:= \sum_m N_m Q_m(T)$
   2b. continue until one reaches the single-node (root) tree
   2c. this produces a sequence of subtrees which contains T_α.

3. Estimation of α is achieved by five- or tenfold cross-validation: $\hat{\alpha}$ minimizes the cross-validated sum of squares.
4. Final tree is $T_{\hat{\alpha}}$.

Cost-complexity pruning

1. Grow a large tree T_0, stopping the splitting process only when some minimum node size (say 5) is reached.
2. For each α, find the subtree T_α that minimizes $C_\alpha(T)$:

$$T_\alpha = \arg\min_{T \subseteq T_0} \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|,$$

   using weakest-link pruning[1]:

   2a. successively collapse the internal node that produces the smallest per-node increase in the fit $:= \sum_m N_m Q_m(T)$
   2b. continue until one reaches the single-node (root) tree
   2c. this produces a sequence of subtrees which contains T_α.

3. Estimation of α is achieved by five- or tenfold cross-validation: $\hat{\alpha}$ minimizes the cross-validated sum of squares.
4. Final tree is $T_{\hat{\alpha}}$.

Menu

Classification trees

In classification trees, each leave (terminal node) is associated with a label.

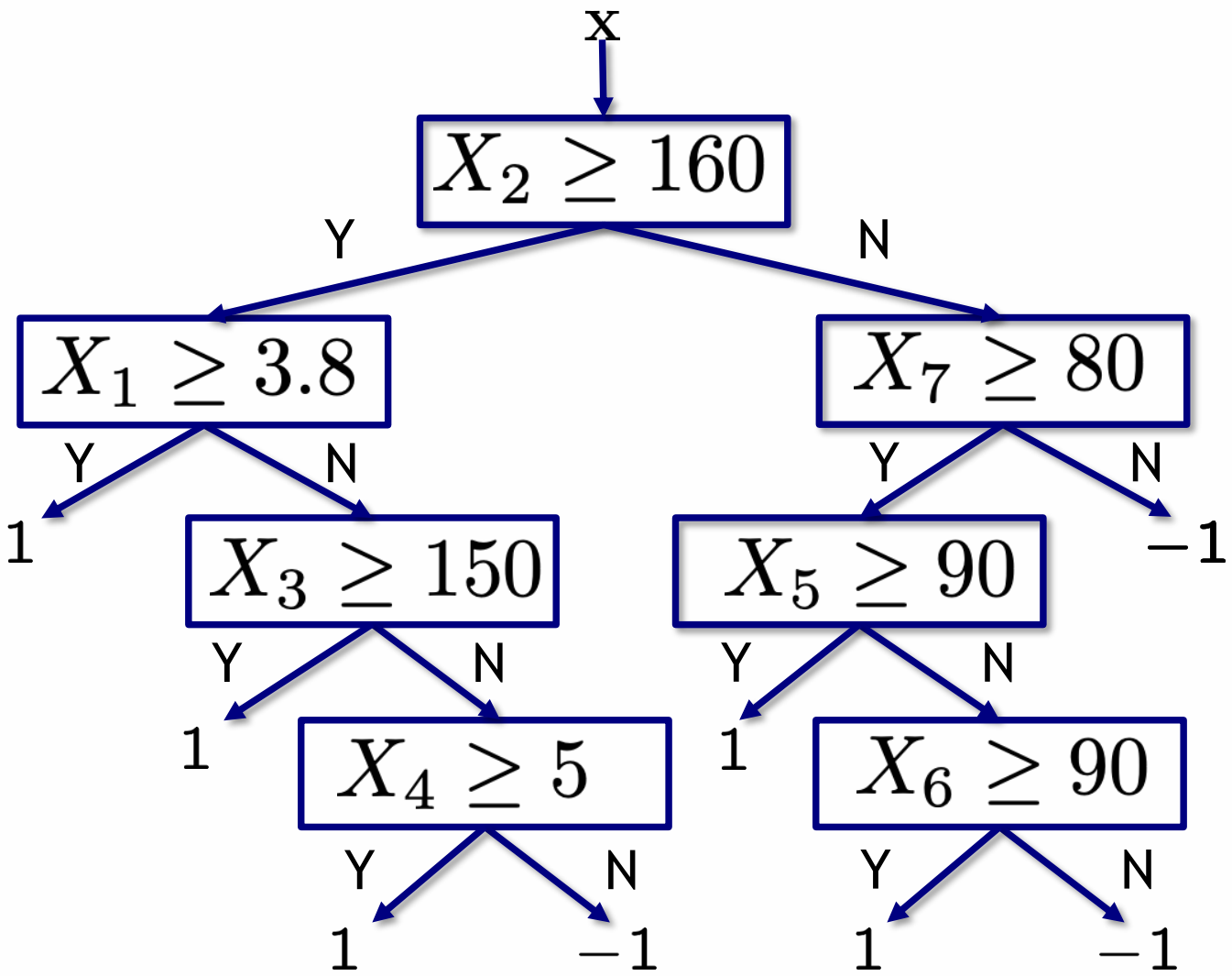
Consider for example, a data with seven inputs (features):

1. $X_1 =$ Undergraduate GPA (4.0 scale)
2. $X_2 =$ Quantitative GRE score (130-170)
3. $X_3 =$ Verbal GRE score (130-170)
4. $X_4 =$ Analytical Writing GRE score (0-6)
5. $X_5 =$ Undergraduate institution reputation (0-100)
6. $X_6 =$ Statement of purpose evaluation (0-100)
7. $X_7 =$ Letter of recommendation evaluation (0-100)

and where the output is:

$$Y = \begin{cases} 1, & \text{Student was admitted} \\ -1, & \text{Student was not admitted} \end{cases}$$

Example classification tree



Node impurity for classification trees

- Only changes in the tree algorithm pertain to the criteria for splitting nodes and pruning the tree.

$\Rightarrow$ Node impurity measure $Q_m(T)$ needs to be redefined.

- Let N_m denote the # of observations in the terminal node m that represents the region R_m.
- Let $(\hat{p}_{m,1}, \ldots, \hat{p}_{m,K})$ denote the frequencies:

$$\hat{p}_{m,k} = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} 1_{\{y_i = k\}} := \begin{array}{l} \text{proportion of class } k \\ \text{observations in node } m \end{array}.$$

- We classify the observations in node m to class

$$k(m) = \arg\max_k \hat{p}_{m,k},$$

i.e., the majority class in node m.

On node impurity measure

- Idea: we want children to be more "pure" than the parent.
- One option is to use misclassification error $\text{Err}(m) = 1 - \hat{p}_{m,k(m)}$ as $Q_m(T)$.
- But is turns out to be not be good as illustrated below:

- Parent node is 80%+, and 20% −
  $\Rightarrow$ Error = 20%.
- Child nodes are (90 %+, 10%−) , (70 %+, 30%−)
  $\Rightarrow$ Errors 10% and 30%, average error 20%

$\Rightarrow$ No improvement!

Different measures $Q_m(T)$ of node impurity

- Misclassification error is not concave and non-differentiable.
- Impurity measures (e.g., Gini index and entropy,) which are differentiable and concave are better suited for numerical optimization.
- They are also more sensitive to changes in the node probabilities

Misclassification: $\text{Err}(m) = 1 - \hat{p}_{m,k(m)}$

Gini index: $G(m) = \sum_{k \neq k'} \hat{p}_{m,k}\hat{p}_{m,k'} = 1 - \sum_{k=1}^{K} \hat{p}_{m,k}^2$

- maximized when $\hat{p}_{m,k} = 1/K$ with value $1 - 1/K$
- minimized when all cases belong to a single class.

Entropy/deviance: $H(m) = -\sum_{k=1}^{K} \hat{p}_{m,k} \log \hat{p}_{m,k}$

- maximized when $\hat{p}_{m,k} = 1/K$ with value $\log K$.
- minimized when one class has no cases in it.

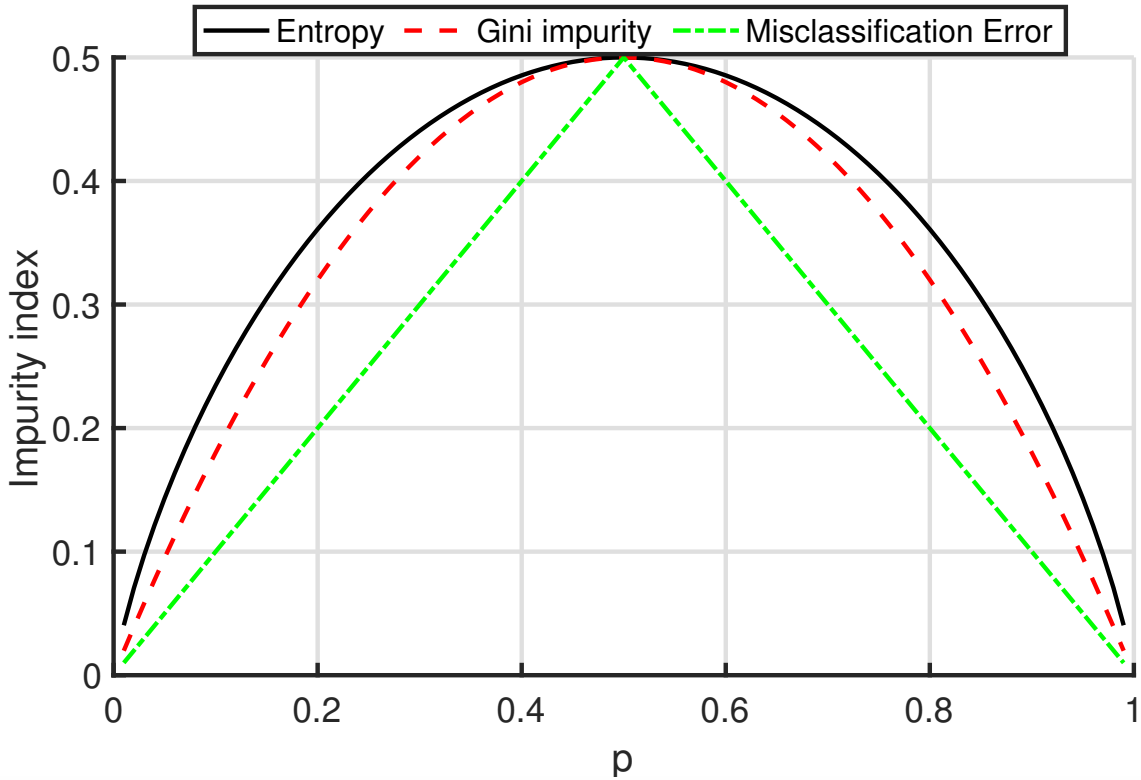
Different measures $Q_m(T)$ of node impurity

Consider binary class problem. Let p denote the proportion of $+$ observations in the terminal node.

Misclassification:
$\text{Err}(m) = \min(p, 1-p)$

Gini:
$G(m) = 2p(1-p)$

Entropy:
$H(m) = -p\log p - (1-p)\log(1-p)$



Note: entropy has been scaled by $1/2$

Menu

Ensemble methods

- The partition of an binary decision tree is
  - **rough** (non-smooth), regardless of how large the tree is grown.
  - **unstable** (or **non-robust**), partition is learned with a lot of variance and a small change in inputs can lead to large change in output.
- Ensemble methods use an *ensemble* (or committee) of classifiers to tackle these deficiencies of the decision trees:
  - Generate a number of classifiers $G_1(\cdot), \ldots, G_B(\cdot)$ based on the same data set but typically using some degree of randomness (e.g., in the selection of the features, or in the selection of the observations from the full data set).

Ensemble methods

- The partition of an binary decision tree is
  - **rough** (non-smooth), regardless of how large the tree is grown.
  - **unstable** (or **non-robust**), partition is learned with a lot of variance and a small change in inputs can lead to large change in output.
- Ensemble methods use an *ensemble* (or committee) of classifiers to tackle these deficiencies of the decision trees:
  1. Generate a number of classifiers $G_1(\cdot), \ldots, G_M(\cdot)$ based on the same data set but typically using some degree of randomness (e.g., in the selection of the features, or in the selection of the observations from the full data set).
  2. Aggregate the classifiers to make the final prediction (take a majority vote).
  3. In the case of ties (e.g., some classes have equal number of votes), then take a random pick of the classes.

Ensemble methods

- The partition of an binary decision tree is
  - **rough** (non-smooth), regardless of how large the tree is grown.
  - **unstable** (or **non-robust**), partition is learned with a lot of variance and a small change in inputs can lead to large change in output.
- Ensemble methods use an *ensemble* (or committee) of classifiers to tackle these deficiencies of the decision trees:
  1. Generate a number of classifiers $G_1(\cdot), \ldots, G_M(\cdot)$ based on the same data set but typically using some degree of randomness (e.g., in the selection of the features, or in the selection of the observations from the full data set).
  2. Aggregate the classifiers to make the final prediction (take a majority vote).
  3. In the case of ties (e.g., some classes have equal number of votes), then take a random pick of the classes.

Ensemble methods

- The partition of an binary decision tree is
  - **rough** (non-smooth), regardless of how large the tree is grown.
  - **unstable** (or **non-robust**), partition is learned with a lot of variance and a small change in inputs can lead to large change in output.
- Ensemble methods use an *ensemble* (or committee) of classifiers to tackle these deficiencies of the decision trees:
  1. Generate a number of classifiers $G_1(\cdot), \ldots, G_M(\cdot)$ based on the same data set but typically using some degree of randomness (e.g., in the selection of the features, or in the selection of the observations from the full data set).
  2. Aggregate the classifiers to make the final prediction (take a majority vote).
  3. In the case of ties (e.g., some classes have equal number of votes), then take a random pick of the classes.

Ensemble methods

- The partition of an binary decision tree is
  - **rough** (non-smooth), regardless of how large the tree is grown.
  - **unstable** (or **non-robust**), partition is learned with a lot of variance and a small change in inputs can lead to large change in output.
- Ensemble methods use an *ensemble* (or committee) of classifiers to tackle these deficiencies of the decision trees:
  1. Generate a number of classifiers $G_1(\cdot), \ldots, G_M(\cdot)$ based on the same data set but typically using some degree of randomness (e.g., in the selection of the features, or in the selection of the observations from the full data set).
  2. Aggregate the classifiers to make the final prediction (take a majority vote).
  3. In the case of ties (e.g., some classes have equal number of votes), then take a random pick of the classes.

Bagging

- Bagging is short for bootstrap aggregation
- A bootstrap sample $\mathcal{T}^* = \{(y_i^*, \mathbf{x}_i^*)\}_{i=1}^N$ is formed by resampling **with replacement** from the original training data $\mathcal{T} = \{(y_i, \mathbf{x}_i)\}_{i=1}^N$.
- Bootstrap sample is generated as follows:
  - Let I_b be a list of size N obtained by sampling from $\{1, \ldots, N\}$ with replacement.
  - A boostrap sample is then $\mathcal{T}^* = \{(y_i, \mathbf{x}_i)\}_{i \in I_b}$

Bootstrap (bu:tstræp)

1. A loop of leather or cloth sewn at the top rear, or sometimes on each side of a boot to facilitate pulling it on.
2. pull oneself up by ones (own) bootstraps, to help oneself without the aid of others ; use ones own resources.

Bagging (cont'd)

- Generate $\mathcal{T}_b^*$, $b = 1, \ldots, B$, such bootstrap samples.
- Let G_b be the classifier when applied to the boostrap sample $\mathcal{T}_b^*$.
- The bagging classifier is then a majority vote over $G_1, \ldots, G_B$:
  - *regression*: $\hat{f}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} G_b(\mathbf{x})$ is the prediction of $\mathbf{x} \in \mathbb{R}^p$.
  - *classification*: Predicted class label for $\mathbf{x} \in \mathbb{R}^p$ is determined using majority voting:

$$\hat{G}(\mathbf{x}) = \arg\max_k \sum_{b=1}^{B} 1_{\{G_b(\mathbf{x})=k\}}$$

or mean probability:

$$\hat{G}(\mathbf{x}) = \arg\max_k \sum_{b=1}^{B} \hat{\Pr}_b(Y = k \mid X = \mathbf{x})$$

where $p^{(b)}(\mathbf{x}) = \hat{\Pr}_b(Y = k|X = \mathbf{x})$ is the probability prediction for kth class obtained by $G_b(\cdot)$

Bagging (cont'd)

- Generate $\mathcal{T}_b^*$, $b = 1, \ldots, B$, such bootstrap samples.
- Let G_b be the classifier when applied to the boostrap sample $\mathcal{T}_b^*$.
- The bagging classifier is then a majority vote over $G_1, \ldots, G_B$:
  - *regression*: $\hat{f}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} G_b(\mathbf{x})$ is the prediction of $\mathbf{x} \in \mathbb{R}^p$.
  - *classification*: Predicted class label for $\mathbf{x} \in \mathbb{R}^p$ is determined using majority voting:

$$\hat{G}(\mathbf{x}) = \arg\max_k \sum_{b=1}^{B} 1_{\{G_b(\mathbf{x})=k\}}$$

or mean probability:

$$\hat{G}(\mathbf{x}) = \arg\max_k \sum_{b=1}^{B} \hat{\Pr}_b(Y = k \mid X = \mathbf{x})$$

where $p^{(b)}(\mathbf{x}) = \hat{\Pr}_b(Y = k|X = \mathbf{x})$ is the probability prediction for kth class obtained by $G_b(\cdot)$

Bagging (cont'd)

- Generate $\mathcal{T}_b^*$, $b = 1, \ldots, B$, such bootstrap samples.
- Let G_b be the classifier when applied to the boostrap sample $\mathcal{T}_b^*$.
- The bagging classifier is then a majority vote over $G_1, \ldots, G_B$:
  - *regression*: $\hat{f}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} G_b(\mathbf{x})$ is the prediction of $\mathbf{x} \in \mathbb{R}^p$.
  - *classification*: Predicted class label for $\mathbf{x} \in \mathbb{R}^p$ is determined using majority voting:

$$\hat{G}(\mathbf{x}) = \arg\max_k \sum_{b=1}^{B} 1_{\{G_b(\mathbf{x})=k\}}$$

or mean probability:

$$\hat{G}(\mathbf{x}) = \arg\max_k \sum_{b=1}^{B} \hat{\Pr}_b(Y = k \mid X = \mathbf{x})$$

where $p^{(b)}(\mathbf{x}) = \hat{\Pr}_b(Y = k|X = \mathbf{x})$ is the probability prediction for kth class obtained by $G_b(\cdot)$

Algorithm 3.1: `Bagging` algorithm using trees

Input : $\mathcal{T} = \{(y_i, \mathbf{x}_i^\top)\}_{i=1}^N$ (data), B (# of bootstrap samples)
Output : Ensemble of trees $\{T_b\}_{b=1}^B$

for $b = 1$ **to** B **do**
  Draw a bootstrap sample $\mathcal{T}_b^*$ of size N from the training data.
  Build a tree $T_b(\cdot)$ on $\mathcal{T}_b^*$

`// To make a prediction at a new point x:`
if *classification task* **then**
  if *not majority voting* **then**
    $\hat{G}(\mathbf{x}) = k$ having largest mean probability $\sum_{b=1}^B p_k^{(b)}(\mathbf{x})$
    `//` above $p^{(b)}(\mathbf{x}) = \hat{\Pr}_b(Y = k|X = \mathbf{x})$ `based on` b`th tree` $T_b(\cdot)$
  else
    $\hat{G}(\mathbf{x}) =$ majority vote over $\{T_b(\mathbf{x})\}_{b=1}^B$
else
  $\hat{f}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^B T_b(\mathbf{x})$.

Out-of-bag (OOB) error rate

- We can use cases not included in the Bootstrap sample (out-of-bag (OOB) observations) as a test set.
- For each observation $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, construct its bagging predictor by averaging only those trees T_b corresponding to bootstrap samples $\mathcal{T}_b^*$ in which $\mathbf{z}_i$ did not appear.
- Averaging means averaging the probability predictions or the class predictions in the case of majority voting.
- If an observation has no prediction yet (which can happen only in the beginning of iterations) one assigns the observation randomly to one of the classes.
- Ties can occur when majority voting is used; in such case one assigns the observation randomly to one of the classes.
- The OOB error estimate is almost identical to that obtained by N-fold cross-validation.
- Once the OOB error stabilizes, the training can be terminated.

Out-of-bag (OOB) error rate

- We can use cases not included in the Bootstrap sample (out-of-bag (OOB) observations) as a test set.
- For each observation $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, construct its bagging predictor by averaging only those trees T_b corresponding to bootstrap samples $\mathcal{T}_b^*$ in which $\mathbf{z}_i$ did not appear.
- Averaging means averaging the probability predictions or the class predictions in the case of majority voting.
- If an observation has no prediction yet (which can happen only in the beginning of iterations) one assigns the observation randomly to one of the classes.
- Ties can occur when majority voting is used; in such case one assigns the observation randomly to one of the classes.
- The OOB error estimate is almost identical to that obtained by N-fold cross-validation.
- Once the OOB error stabilizes, the training can be terminated.

Out-of-bag (OOB) error rate

- We can use cases not included in the Bootstrap sample (out-of-bag (OOB) observations) as a test set.
- For each observation $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, construct its bagging predictor by averaging only those trees T_b corresponding to bootstrap samples $\mathcal{T}_b^*$ in which $\mathbf{z}_i$ did not appear.
- Averaging means averaging the probability predictions or the class predictions in the case of majority voting.
- If an observation has no prediction yet (which can happen only in the beginning of iterations) one assigns the observation randomly to one of the classes.
- Ties can occur when majority voting is used; in such case one assigns the observation randomly to one of the classes.
- The OOB error estimate is almost identical to that obtained by N-fold cross-validation.
- Once the OOB error stabilizes, the training can be terminated.

Out-of-bag (OOB) error rate

- We can use cases not included in the Bootstrap sample (out-of-bag (OOB) observations) as a test set.
- For each observation $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, construct its bagging predictor by averaging only those trees T_b corresponding to bootstrap samples $\mathcal{T}_b^*$ in which $\mathbf{z}_i$ did not appear.
- Averaging means averaging the probability predictions or the class predictions in the case of majority voting.
- If an observation has no prediction yet (which can happen only in the beginning of iterations) one assigns the observation randomly to one of the classes.
- Ties can occur when majority voting is used; in such case one assigns the observation randomly to one of the classes.
- The OOB error estimate is almost identical to that obtained by N-fold cross-validation.
- Once the OOB error stabilizes, the training can be terminated.

Out-of-bag (OOB) error rate

- We can use cases not included in the Bootstrap sample (out-of-bag (OOB) observations) as a test set.
- For each observation $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, construct its bagging predictor by averaging only those trees T_b corresponding to bootstrap samples $\mathcal{T}_b^*$ in which $\mathbf{z}_i$ did not appear.
- Averaging means averaging the probability predictions or the class predictions in the case of majority voting.
- If an observation has no prediction yet (which can happen only in the beginning of iterations) one assigns the observation randomly to one of the classes.
- Ties can occur when majority voting is used; in such case one assigns the observation randomly to one of the classes.
- The OOB error estimate is almost identical to that obtained by N-fold cross-validation.
- Once the OOB error stabilizes, the training can be terminated.

Out-of-bag (OOB) error rate

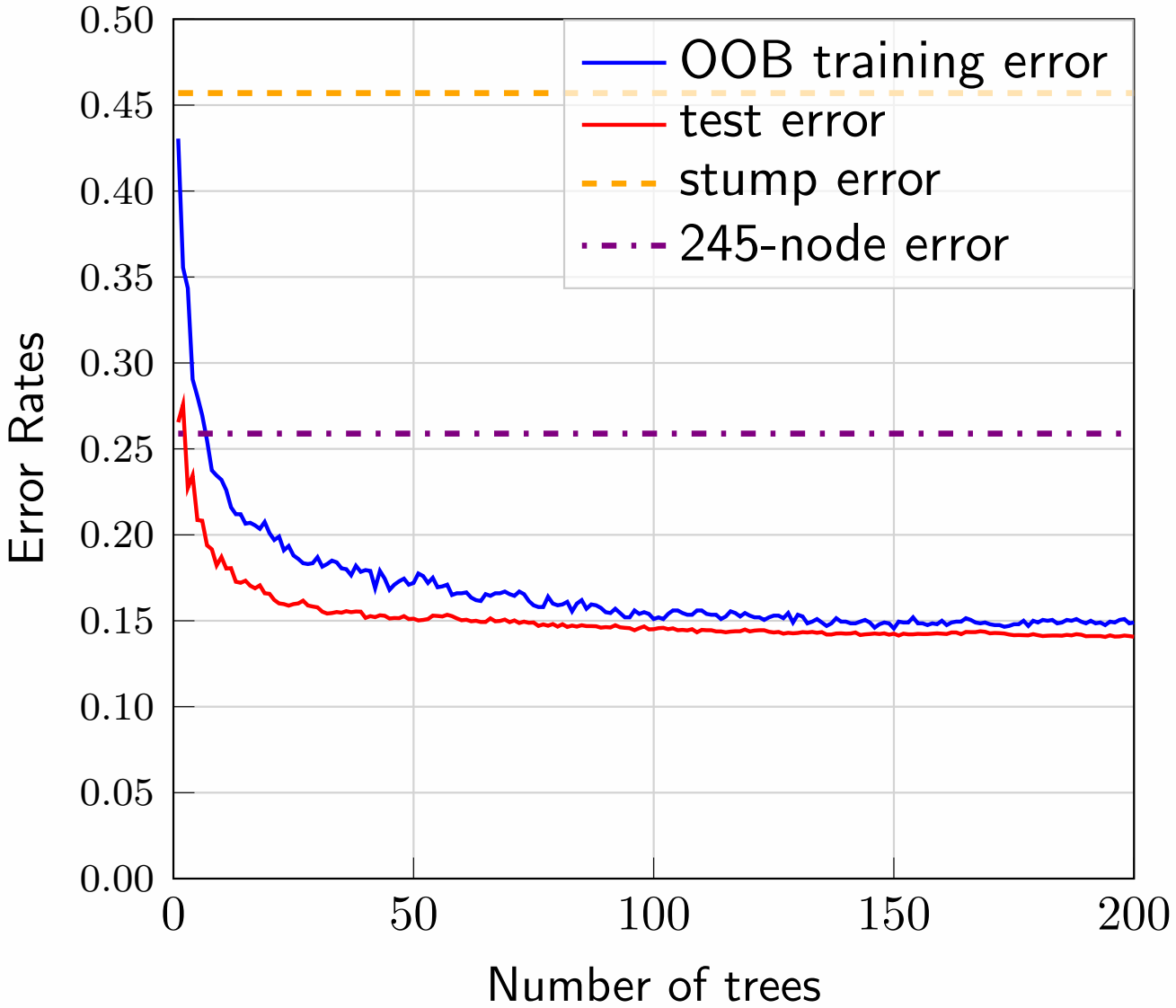
- We can use cases not included in the Bootstrap sample (out-of-bag (OOB) observations) as a test set.
- For each observation $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, construct its bagging predictor by averaging only those trees T_b corresponding to bootstrap samples $\mathcal{T}_b^*$ in which $\mathbf{z}_i$ did not appear.
- Averaging means averaging the probability predictions or the class predictions in the case of majority voting.
- If an observation has no prediction yet (which can happen only in the beginning of iterations) one assigns the observation randomly to one of the classes.
- Ties can occur when majority voting is used; in such case one assigns the observation randomly to one of the classes.
- The OOB error estimate is almost identical to that obtained by N-fold cross-validation.
- Once the OOB error stabilizes, the training can be terminated.

Out-of-bag (OOB) error rate

- We can use cases not included in the Bootstrap sample (out-of-bag (OOB) observations) as a test set.
- For each observation $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, construct its bagging predictor by averaging only those trees T_b corresponding to bootstrap samples $\mathcal{T}_b^*$ in which $\mathbf{z}_i$ did not appear.
- Averaging means averaging the probability predictions or the class predictions in the case of majority voting.
- If an observation has no prediction yet (which can happen only in the beginning of iterations) one assigns the observation randomly to one of the classes.
- Ties can occur when majority voting is used; in such case one assigns the observation randomly to one of the classes.
- The OOB error estimate is almost identical to that obtained by N-fold cross-validation.
- Once the OOB error stabilizes, the training can be terminated.

Example 3.1

Binary ($K = 2$) classification problem on synthetic (simulated) data of $p = 10$ features.



Menu

Random forests

- Random forests = bagging using decision trees with random feature selection:
  1. Generate classifiers by choosing random subsets of d features and construct a decision tree on just the selected features.
     Random subsets are considered *at each internal node* so that the search space for each split is smaller.
  2. Combine the approach with bagging.
- The obtained partitions are less correlated, and hence the obtained final aggregate prediction has reduced variance.
- Due to the reduced space at each split, the individual trees are built much faster than in bagging.
- Rule of thumb: use $d = \sqrt{p}$.

Random forests

- Random forests = bagging using decision trees with random feature selection:
  1. Generate classifiers by choosing random subsets of d features and construct a decision tree on just the selected features.
     Random subsets are considered *at each internal node* so that the search space for each split is smaller.
  2. Combine the approach with bagging.
- The obtained partitions are less correlated, and hence the obtained final aggregate prediction has reduced variance.
- Due to the reduced space at each split, the individual trees are built much faster than in bagging.
- Rule of thumb: use $d = \sqrt{p}$.

Random forests

- Random forests = bagging using decision trees with random feature selection:
  1. Generate classifiers by choosing random subsets of d features and construct a decision tree on just the selected features.
     Random subsets are considered *at each internal node* so that the search space for each split is smaller.
  2. Combine the approach with bagging.
- The obtained partitions are less correlated, and hence the obtained final aggregate prediction has reduced variance.
- Due to the reduced space at each split, the individual trees are built much faster than in bagging.
- Rule of thumb: use $d = \sqrt{p}$.

Random forests

- Random forests = bagging using decision trees with random feature selection:
  1. Generate classifiers by choosing random subsets of d features and construct a decision tree on just the selected features.
     Random subsets are considered *at each internal node* so that the search space for each split is smaller.
  2. Combine the approach with bagging.
- The obtained partitions are less correlated, and hence the obtained final aggregate prediction has reduced variance.
- Due to the reduced space at each split, the individual trees are built much faster than in bagging.
- Rule of thumb: use $d = \sqrt{p}$.

Algorithm 3.2: `RandomForest` algorithm for classification or regression

Input : $\mathcal{T} = \{(y_i, \mathbf{x}_i)\}_{i=1}^N$, B (# of bootstrap samples), d (# of features in each split), n_{min} (minimum node size)

Output : Ensemble of trees $\{T_b\}_{b=1}^B$

for $b = 1$ **to** B **do**

- Draw a bootstrap sample $\mathcal{T}_b^*$ of size N from the training data.
- **for** *each terminal node in tree* **do**
  - Select d variables at random from the p variables.
  - Pick the best variable/split-point among the d variables.
  - Split the node into two daughter nodes on the selected feature until the minimum node size n_{min} is reached

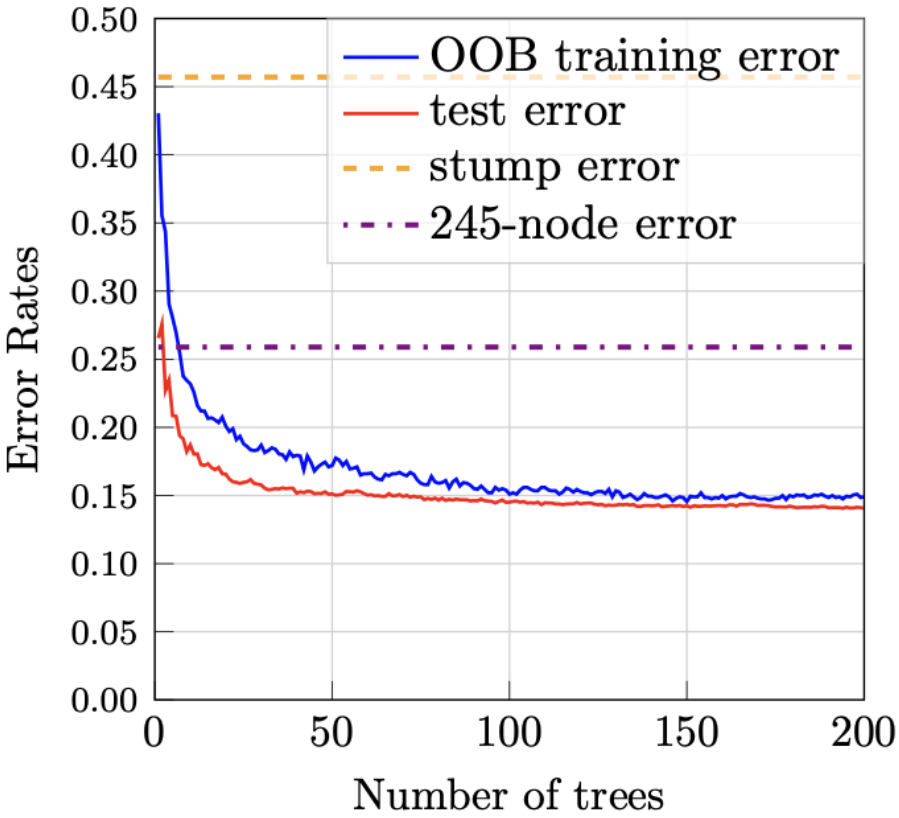
if *classification task* **then**

- **if** *not majority voting* **then**
  - $\hat{G}(\mathbf{x}) = k$ having largest mean probability $\sum_{b=1}^B p_k^{(b)}(\mathbf{x})$
- **else**
  - $\hat{G}(\mathbf{x}) =$ majority vote over $\{T_b(\mathbf{x})\}_{b=1}^B$

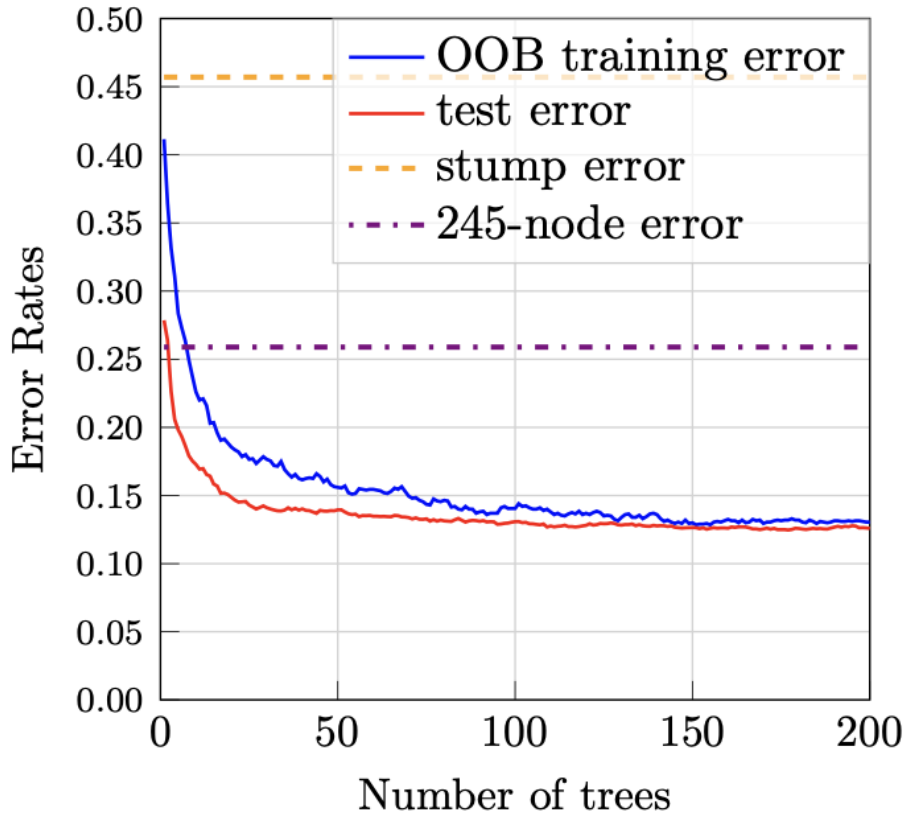
else

- $\hat{f}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^B T_b(\mathbf{x})$.

Example 3.2



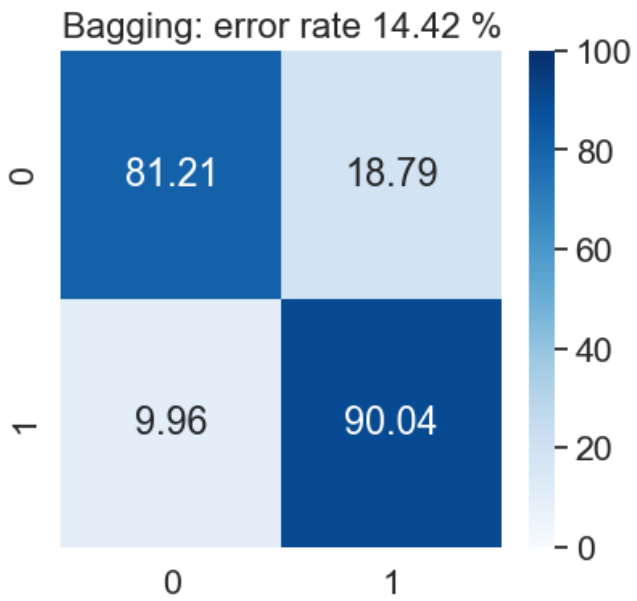
(A) Bagging



(B) Random Forest

Random forest achieves error rate of 12.59% with 200 bags while bagging alone achieved 14.42% test error rate.

Confusion matrices



Menu

General ensemble scheme

$$\text{input } \{(\mathbf{x}_i, y_i)\}_{i=1}^N \longrightarrow \boxed{\text{weak learner}} \longrightarrow G(\cdot)$$

Boosting:

- Applies *sequentially* the weak classification/regression algorithm to repeatedly modified versions of the data,
- Produces a sequence of weak learners $G_m(\mathbf{x})$, $m = 1, 2, \ldots, M$.
- Applicable to a regression or classification algorithm that accepts case weights, e.g.,

$$G = \arg\min_f \sum_{i=1}^N w_i L(y_i, f(\mathbf{x}_i)).$$

- Predictions from all learners are combined through a weighted majority vote.

General boosting ensemble scheme

initialize: $w_i^{(0)} = 1/N,\ i = 1, \ldots, N$

data 1 $\left\{(\mathbf{x}_i, y_i, w_i^{(0)})\right\}_{i=1}^N \longrightarrow$ weak learner $\longrightarrow G_1(\cdot), \alpha_1$

data 2 $\left\{(\mathbf{x}_i, y_i, w_i^{(1)})\right\}_{i=1}^N \longrightarrow$ weak learner $\longrightarrow G_2(\cdot), \alpha_2$

$\vdots$

data M $\left\{(\mathbf{x}_i, y_i, w_i^{(M-1)})\right\}_{i=1}^N \longrightarrow$ weak learner $\longrightarrow G_M(\cdot), \alpha_M$

output: Weighted majority vote $\hat{f}(\mathbf{x}) = \sum_{m=1}^M \alpha_m G_m(\mathbf{x})$.

For example, in the case of binary classification problem, the final prediction is:

$$G(x) = \text{sign}(\hat{f}(\mathbf{x})) = \text{sign}\left(\sum_{m=1}^M \alpha_m G_m(\mathbf{x})\right).$$

General boosting ensemble scheme (cont'd)

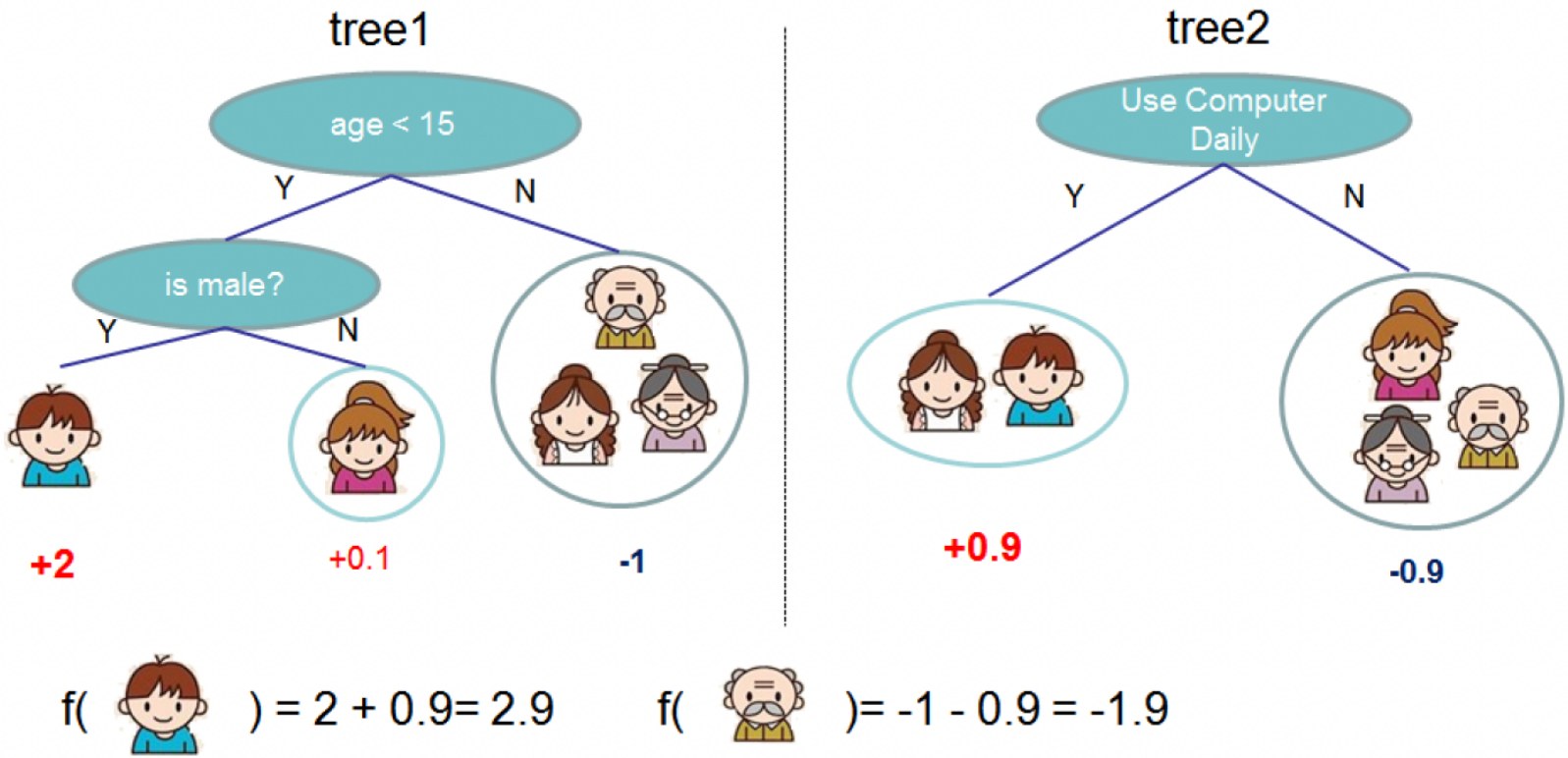


Figure from Chen and Guestrin [2016]

- The final prediction for a given example is the sum of predictions from each tree.
- Classification: sign of the predictor, $\text{sign}(\hat{f}(\mathbf{x}))$, is the output $G(\mathbf{x})$.

General boosting ensemble scheme (cont'd)

- The classifier/regression weights, $\alpha_1, \ldots, \alpha_M$, computed by the boosting algorithm are designed so that more accurate classifier/regressor $G_m(\mathbf{x})$ in the sequence obtains a larger weight and thus has a higher influence on $G(\mathbf{x})$.
- The data weights $w_i^{(m)}$, $i = 1, \ldots, N$ at each boosting iteration m depends on the accuracy of the previous learners.
- Data weights allow the algorithm to focus its attention on those samples that are still incorrectly classified.

Menu

Adaboost

- The AdaBoost algorithm [Freund and Schapire, 1996, 1997] is the most well known boosting algorithm, and originally developed for binary classification problem.
- Here the base classifier $G(\mathbf{x})$ attains values in $\{-1, 1\}$.
- An important tuning parameter is # of weak learners M:
  - AdaBoost (and other boosting methods) are often quite resistant to overfitting when increasing M.
  - Nevertheless, clear overfitting does occur for some datasets, and thus early stopping, i.e., choosing a valid number of iterations M is often necessary.

Algorithm 4.1: AdaBoost.M1 (alias Discrete AdaBoost)

Initialize: $w_i^{(0)} = 1/N,\ i = 1, \ldots, N$

for $m = 1$ **to** M **do**

Fit a classifier $G_m(\mathbf{x}) \in \{-1, 1\}$ to the training data using weights $w_i^{(m-1)}$.

Compute the weighted error rate

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i^{(m-1)} 1_{\{y_i \neq G_m(\mathbf{x}_i)\}}}{\sum_{i=1}^{N} w_i^{(m-1)}}$$

Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.

Update the weights $w_i^{(m)} = w_i^{(m-1)} \cdot \exp\left[\alpha_m \cdot 1_{\{y_i \neq G_m(\mathbf{x}_i)\}}\right]$, $i = 1, \ldots, N$.

Output : $G(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(\mathbf{x})\right)$

Adaboost.M1: key points

1. Initially, $w_i = 1/N$, so at first step one trains the classifier on the data in the usual manner.
2. For each successive iteration $m = 2, 3, \ldots, M$ the observation weights are individually modified and the classification algorithm is reapplied to the weighted observations.
3. At step m, observations that were misclassified by $G_{m-1}(\mathbf{x})$ induced at the previous step have their weights *increased*.
4. Thus, as iterations proceed, observations that are difficult to classify correctly receive ever-increasing influence.

Adaboost.R

- AdaBoost.R (alias Real AdaBoost) [Freund and Schapire, 1996] is a generalization of AdaBoost.M
- It uses real-valued predictions rather than the $\{-1, 1\}$ of Discrete AdaBoost.
- Each weak learner returns a class probability estimate $p_m(\mathbf{x}) = \hat{\Pr}_{\mathbf{w}}(Y = 1 \mid X = \mathbf{x}) \in [0, 1]$ at m^{th} iteration.
- Its contribution to the final classifier is one half the logit-transform of this probability estimate.
- Can be applied for a learner that can compute probabilities of the classes.
- Scikit-learn's `AdaBoostClassifier` uses AdaBoost.R as default. Choose option, `algorithm='SAMME'`, for AdaBoost.M1.

Adaboost.R

- AdaBoost.R (alias Real AdaBoost) [Freund and Schapire, 1996] is a generalization of AdaBoost.M
- It uses real-valued predictions rather than the $\{-1, 1\}$ of Discrete AdaBoost.
- Each weak learner returns a class probability estimate $p_m(\mathbf{x}) = \hat{\Pr}_{\mathbf{w}}(Y = 1 \mid X = \mathbf{x}) \in [0, 1]$ at m^{th} iteration.
- Its contribution to the final classifier is one half the logit-transform of this probability estimate.
- Can be applied for a learner that can compute probabilities of the classes.
- Scikit-learn's `AdaBoostClassifier` uses AdaBoost.R as default. Choose option, `algorithm='SAMME'`, for AdaBoost.M1.

Adaboost.R

- AdaBoost.R (alias Real AdaBoost) [Freund and Schapire, 1996] is a generalization of AdaBoost.M
- It uses real-valued predictions rather than the $\{-1, 1\}$ of Discrete AdaBoost.
- Each weak learner returns a class probability estimate $p_m(\mathbf{x}) = \hat{\Pr}_{\mathbf{w}}(Y = 1 \,|\, X = \mathbf{x}) \in [0, 1]$ at m^{th} iteration.
- Its contribution to the final classifier is one half the logit-transform of this probability estimate.
- Can be applied for a learner that can compute probabilities of the classes.
- Scikit-learn's `AdaBoostClassifier` uses AdaBoost.R as default. Choose option, `algorithm='SAMME'`, for AdaBoost.M1.

Adaboost.R

- AdaBoost.R (alias Real AdaBoost) [Freund and Schapire, 1996] is a generalization of AdaBoost.M
- It uses real-valued predictions rather than the $\{-1, 1\}$ of Discrete AdaBoost.
- Each weak learner returns a class probability estimate $p_m(\mathbf{x}) = \hat{\Pr}_{\mathbf{w}}(Y = 1 \mid X = \mathbf{x}) \in [0, 1]$ at m^{th} iteration.
- Its contribution to the final classifier is one half the logit-transform of this probability estimate.
- Can be applied for a learner that can compute probabilities of the classes.
- Scikit-learn's `AdaBoostClassifier` uses AdaBoost.R as default. Choose option, `algorithm='SAMME'`, for AdaBoost.M1.

Adaboost.R

- AdaBoost.R (alias Real AdaBoost) [Freund and Schapire, 1996] is a generalization of AdaBoost.M
- It uses real-valued predictions rather than the $\{-1, 1\}$ of Discrete AdaBoost.
- Each weak learner returns a class probability estimate $p_m(\mathbf{x}) = \hat{\Pr}_{\mathbf{w}}(Y = 1 \mid X = \mathbf{x}) \in [0, 1]$ at m^{th} iteration.
- Its contribution to the final classifier is one half the logit-transform of this probability estimate.
- Can be applied for a learner that can compute probabilities of the classes.
- Scikit-learn's `AdaBoostClassifier` uses AdaBoost.R as default. Choose option, `algorithm='SAMME'`, for AdaBoost.M1.

Adaboost.R

- AdaBoost.R (alias Real AdaBoost) [Freund and Schapire, 1996] is a generalization of AdaBoost.M
- It uses real-valued predictions rather than the $\{-1, 1\}$ of Discrete AdaBoost.
- Each weak learner returns a class probability estimate $p_m(\mathbf{x}) = \hat{\Pr}_{\mathbf{w}}(Y = 1 \,|\, X = \mathbf{x}) \in [0, 1]$ at m^{th} iteration.
- Its contribution to the final classifier is one half the logit-transform of this probability estimate.
- Can be applied for a learner that can compute probabilities of the classes.
- Scikit-learn's `AdaBoostClassifier` uses AdaBoost.R as default. Choose option, `algorithm='SAMME'`, for AdaBoost.M1.

Algorithm 4.2: AdaBoost.R (or Real AdaBoost)

Initialize: $w_i^{(0)} = 1/N,\ i = 1, \ldots, N$

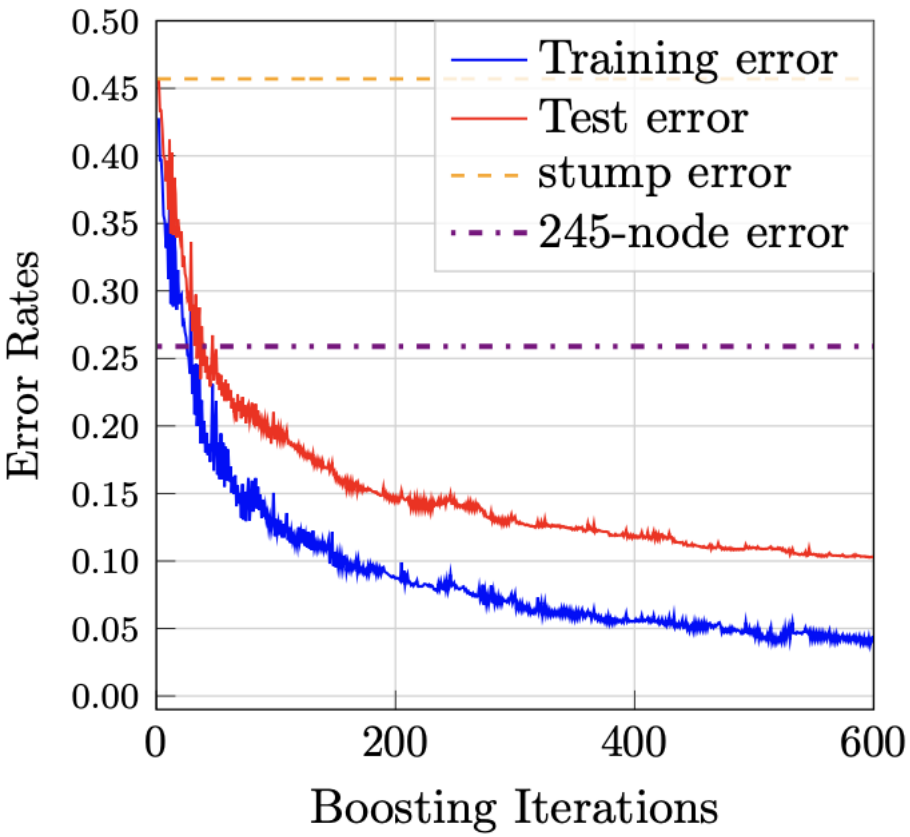
for $m = 1$ **to** M **do**

- Fit a classifier $G_m(\mathbf{x})$ to obtain a class probability estimate $p_m(\mathbf{x}) = \hat{\Pr}_{\mathbf{w}}(Y = 1|\mathbf{x}) \in [0, 1]$, using weights $w_i^{(m-1)}$ on the training data
- Compute $f_m(\mathbf{x}) = \frac{1}{2} \log p_m(\mathbf{x})/(1 - p_m(\mathbf{x})) \in \mathbb{R}$.
- Compute $w_i^{(m)} = w_i^{(m-1)} \exp\{-y_i f_m(\mathbf{x}_i)\},\ i = 1, 2, \ldots, N$ and renormalize so that $\sum_i w_i^{(m)} = 1$.

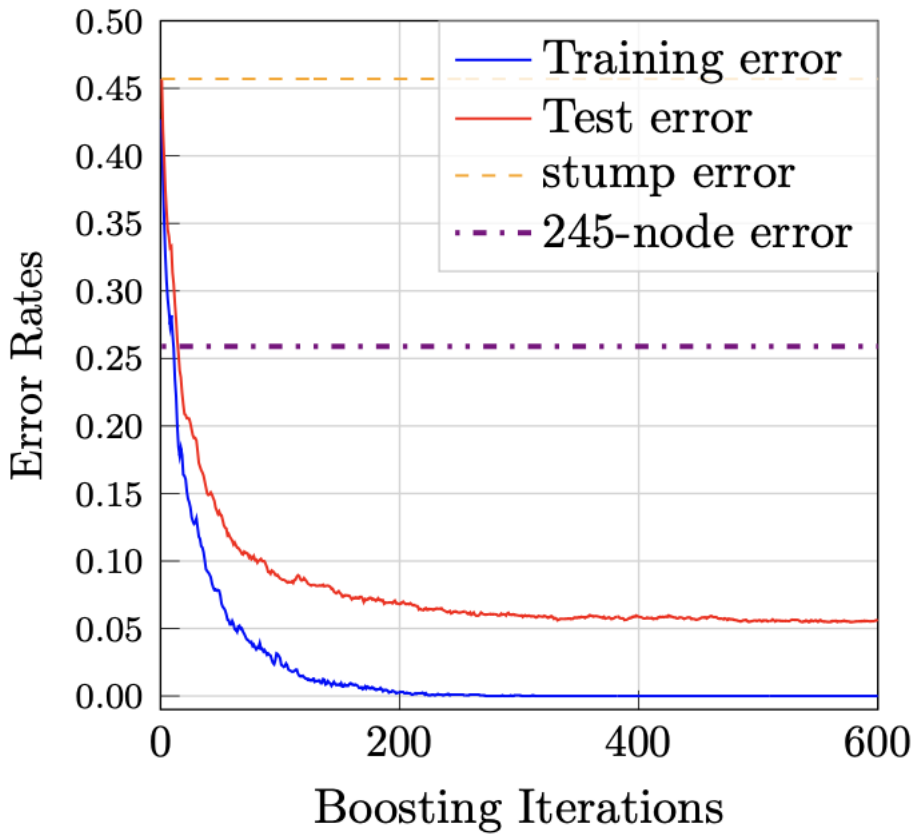
Output : $G(\mathbf{x}) = \text{sign}\left(\hat{f}(\mathbf{x})\right)$ and $\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} f_m(\mathbf{x})$

Example 4.1

Binary ($K = 2$) classification problem on synthetic (simulated) nested-spheres data of $p = 10$ features.



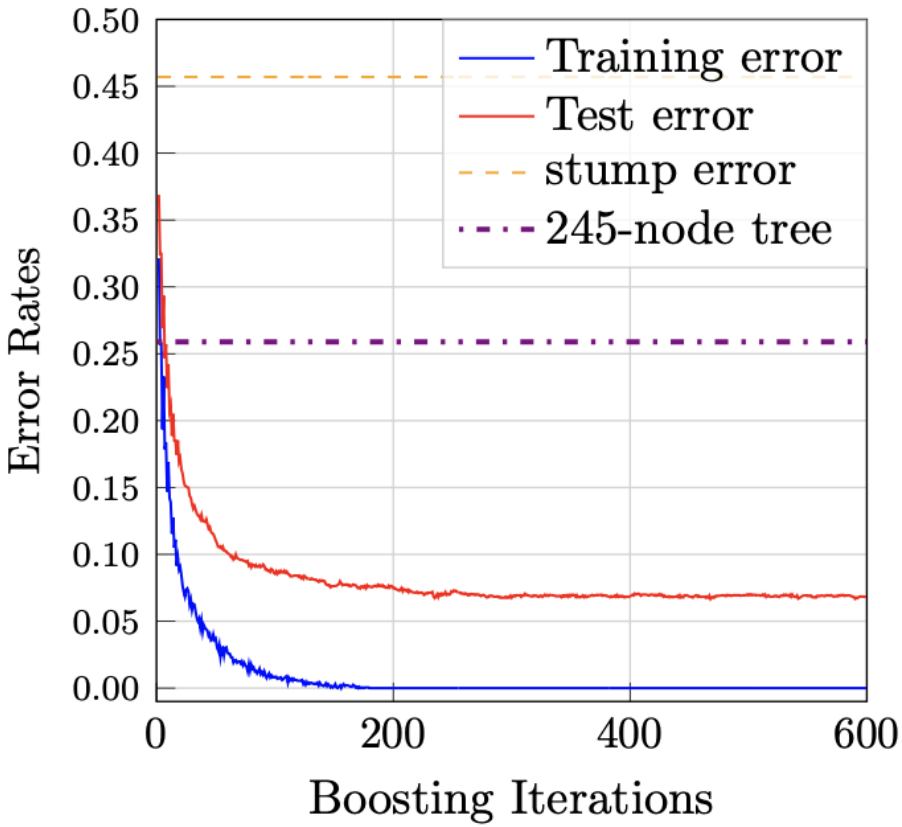
(A) Adaboost.M1 with stumps



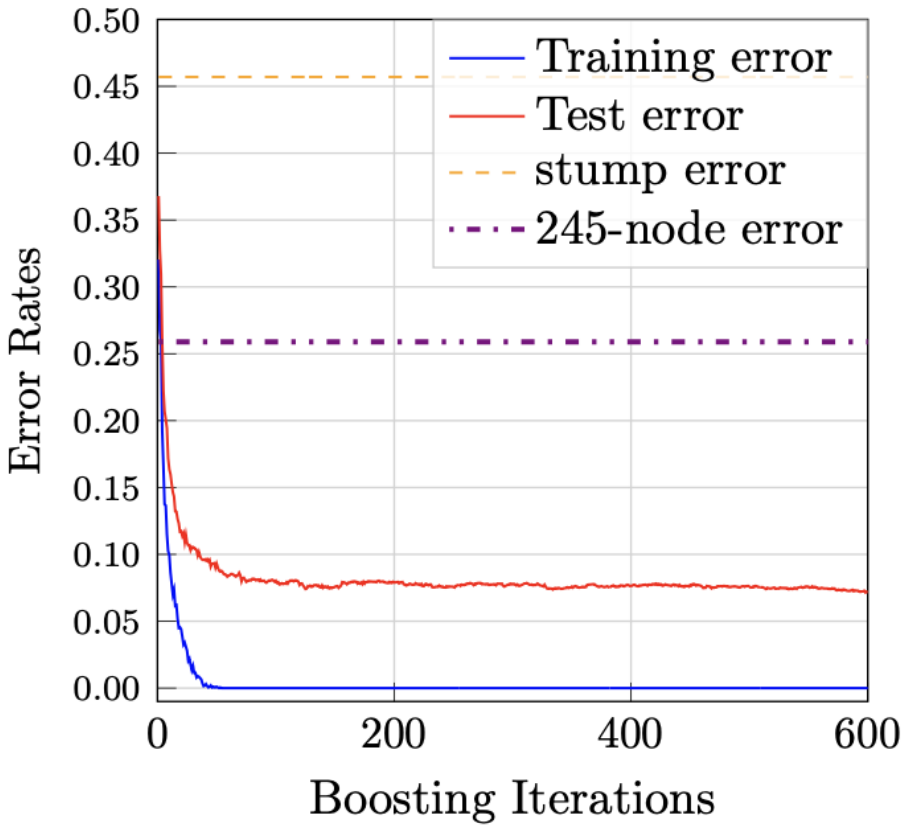
(B) AdaBoost.R with stumps

Example 4.1 (continued)

Binary ($K = 2$) classification problem on synthetic (simulated) nested-spheres data of $p = 10$ features.



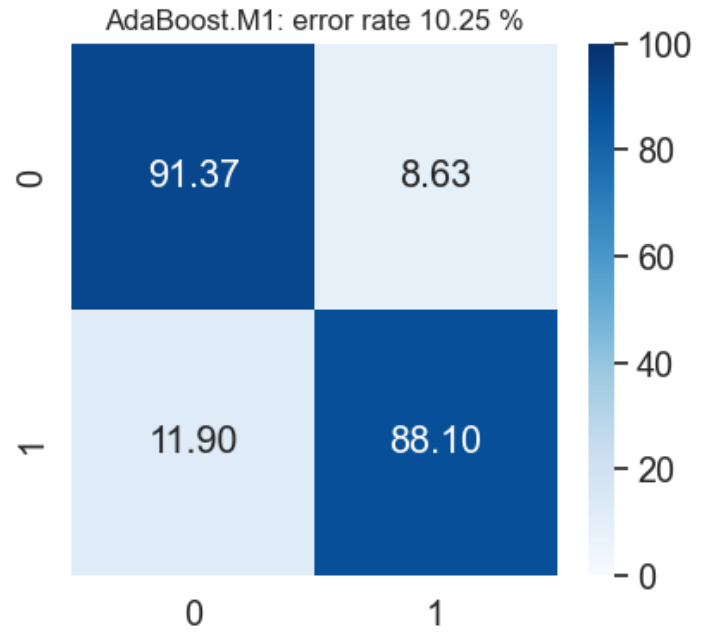
(c) AdaBoost.M1 with 8-node tree

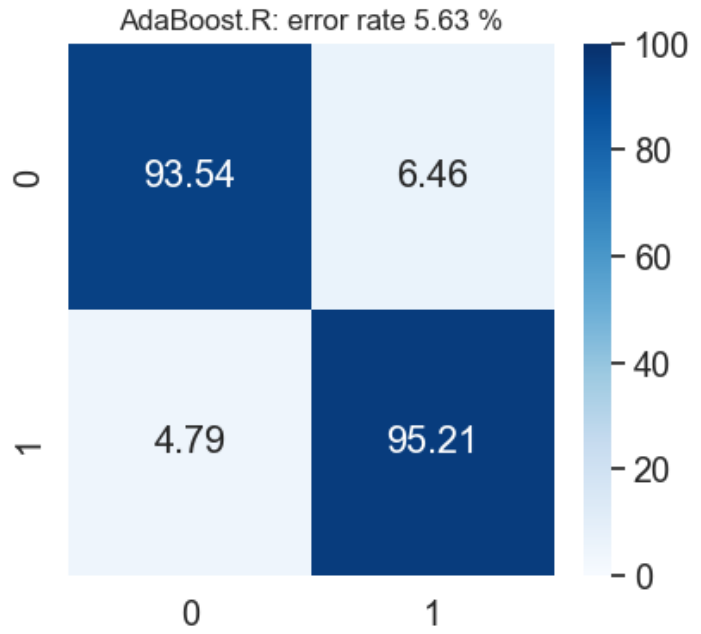


(d) AdaBoost.R with 8-node tree

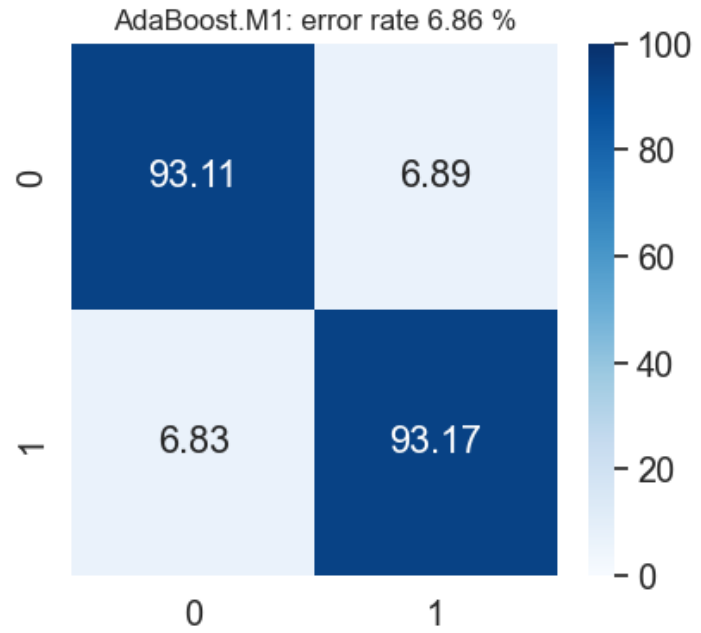
Example 4.1 (continued)

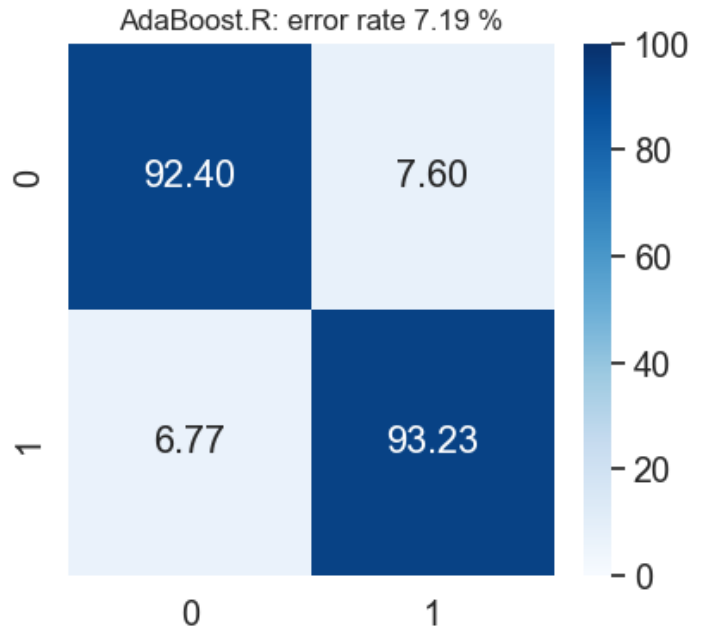
Using stumps





Using 8-node trees





Menu

Forward Stagewise Additive Modeling (FSAM)

Algorithm 4.3: Forward Stagewise Additive Modeling

Initialize: $f_0(\mathbf{x}) = 0$

for $m = 1$ **to** M **do**

Compute

$$(\beta_m, \boldsymbol{\gamma}_m) = \arg\min_{\beta,\boldsymbol{\gamma}} \sum_{i=1}^{N} L\big(y_i, f_{m-1}(\mathbf{x}_i) + \beta b(\mathbf{x}_i; \boldsymbol{\gamma})\big)$$

Set $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \beta_m b(\mathbf{x}; \boldsymbol{\gamma}_m)$.

- β_m, $m = 1, 2, \ldots, M$ are expansion coefficients
- $b(\mathbf{x}; \boldsymbol{\gamma}) \in \mathbb{R}$ are "basis" functions of $\mathbf{x}$, characterized by a set of parameters $\boldsymbol{\gamma}$.
- $L(\cdot, \cdot)$ is the loss function, e.g., $L(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$.

Exponential Loss + FSAM = AdaBoost.M1

- AdaBoost.M1 can be derived using FSAM algorithm:
  - Choosing $L(\cdot, \cdot)$ as the exponential loss

$$L(y, f(\mathbf{x})) = \exp(-yf(\mathbf{x}))$$

  - Base learners $G_m(\mathbf{x}) \in \{-1, 1\}$ taking the role of basis functions $b(\mathbf{x}; \boldsymbol{\gamma})$.
- Thus one solves:

$$\begin{aligned}(\beta_m, G_m) &= \arg\min_{\beta, G} \sum_{i=1}^{N} \exp\big\{ - y_i \big(f_{m-1}(\mathbf{x}_i) + \beta G(\mathbf{x}_i) \big) \big\} \\ &= \arg\min_{\beta, G} \sum_{i=1}^{N} w_i^{(m)} \exp\big(- \beta y_i G(\mathbf{x}_i) \big),\end{aligned}$$

where

$$w_i^{(m)} = \exp\big(- y_i \, f_{m-1}(\mathbf{x}_i) \big)$$

can be regarded as a weight that is applied to each observation.

Class prediction probabilities

- Recall that the optimum predictor function $f^*(\mathbf{x})$ that mininizes the exponential loss is $1/2t$ log odds.
- Additive expansion $\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x})$ can be interpreted as an estimate of log-odds of $p(\mathbf{x}) = \Pr(Y = 1|\mathbf{x})$, so $\hat{f}(\mathbf{x}) = \sigma^{-1}(\hat{p}(\mathbf{x}))$.
- Note: it is not $\frac{1}{2}\times$ times log-odds since the actual multiplier is $\beta_m = \alpha/2$ (see lecture notes).
- This allows one to define probability estimates as

$$\hat{p}(\mathbf{x}) = \frac{1}{1 + e^{-\hat{f}(\mathbf{x})}}.$$

where

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x}) / \sum_{m=1}^{M} \alpha_m$$

is weighted mean predictions in the ensemble.

- This is what `.predict_proba` returns for `AdaBoostClassifier` with option `algorithm='SAMME'`.

Class prediction probabilities

- Recall that the optimum predictor function $f^*(\mathbf{x})$ that mininizes the exponential loss is $1/2t$ log odds.
- Additive expansion $\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x})$ can be interpreted as an estimate of log-odds of $p(\mathbf{x}) = \Pr(Y = 1|\mathbf{x})$, so $\hat{f}(\mathbf{x}) = \sigma^{-1}(\hat{p}(\mathbf{x}))$.
- Note: it is not $\frac{1}{2}\times$ times log-odds since the actual multiplier is $\beta_m = \alpha/2$ (see lecture notes).
- This allows one to define probability estimates as

$$\hat{p}(\mathbf{x}) = \frac{1}{1 + e^{-\hat{f}(\mathbf{x})}}.$$

where

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x}) / \sum_{m=1}^{M} \alpha_m$$

is weighted mean predictions in the ensemble.

- This is what `.predict_proba` returns for `AdaBoostClassifier` with option `algorithm='SAMME'`.

Class prediction probabilities

- Recall that the optimum predictor function $f^*(\mathbf{x})$ that mininizes the exponential loss is $1/2t$ log odds.
- Additive expansion $\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x})$ can be interpreted as an estimate of log-odds of $p(\mathbf{x}) = \Pr(Y = 1|\mathbf{x})$, so $\hat{f}(\mathbf{x}) = \sigma^{-1}(\hat{p}(\mathbf{x}))$.
- Note: it is not $\frac{1}{2}\times$ times log-odds since the actual multiplier is $\beta_m = \alpha/2$ (see lecture notes).
- This allows one to define probability estimates as

$$\hat{p}(\mathbf{x}) = \frac{1}{1 + e^{-\hat{f}(\mathbf{x})}}.$$

where

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x}) / \sum_{m=1}^{M} \alpha_m$$

is weighted mean predictions in the ensemble.

- This is what `.predict_proba` returns for `AdaBoostClassifier` with option `algorithm='SAMME'`.

Class prediction probabilities

- Recall that the optimum predictor function $f^*(\mathbf{x})$ that mininizes the exponential loss is $1/2t$ log odds.
- Additive expansion $\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x})$ can be interpreted as an estimate of log-odds of $p(\mathbf{x}) = \Pr(Y = 1|\mathbf{x})$, so $\hat{f}(\mathbf{x}) = \sigma^{-1}(\hat{p}(\mathbf{x}))$.
- Note: it is not $\frac{1}{2}\times$ times log-odds since the actual multiplier is $\beta_m = \alpha/2$ (see lecture notes).
- This allows one to define probability estimates as

$$\hat{p}(\mathbf{x}) = \frac{1}{1 + e^{-\hat{f}(\mathbf{x})}}.$$

where

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x}) / \sum_{m=1}^{M} \alpha_m$$

is weighted mean predictions in the ensemble.

- This is what `.predict_proba` returns for `AdaBoostClassifier` with option `algorithm='SAMME'`.

Class prediction probabilities

- Recall that the optimum predictor function $f^*(\mathbf{x})$ that mininizes the exponential loss is $1/2t$ log odds.
- Additive expansion $\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x})$ can be interpreted as an estimate of log-odds of $p(\mathbf{x}) = \Pr(Y = 1|\mathbf{x})$, so $\hat{f}(\mathbf{x}) = \sigma^{-1}(\hat{p}(\mathbf{x}))$.
- Note: it is not $\frac{1}{2}\times$ times log-odds since the actual multiplier is $\beta_m = \alpha/2$ (see lecture notes).
- This allows one to define probability estimates as

$$\hat{p}(\mathbf{x}) = \frac{1}{1 + e^{-\hat{f}(\mathbf{x})}}.$$

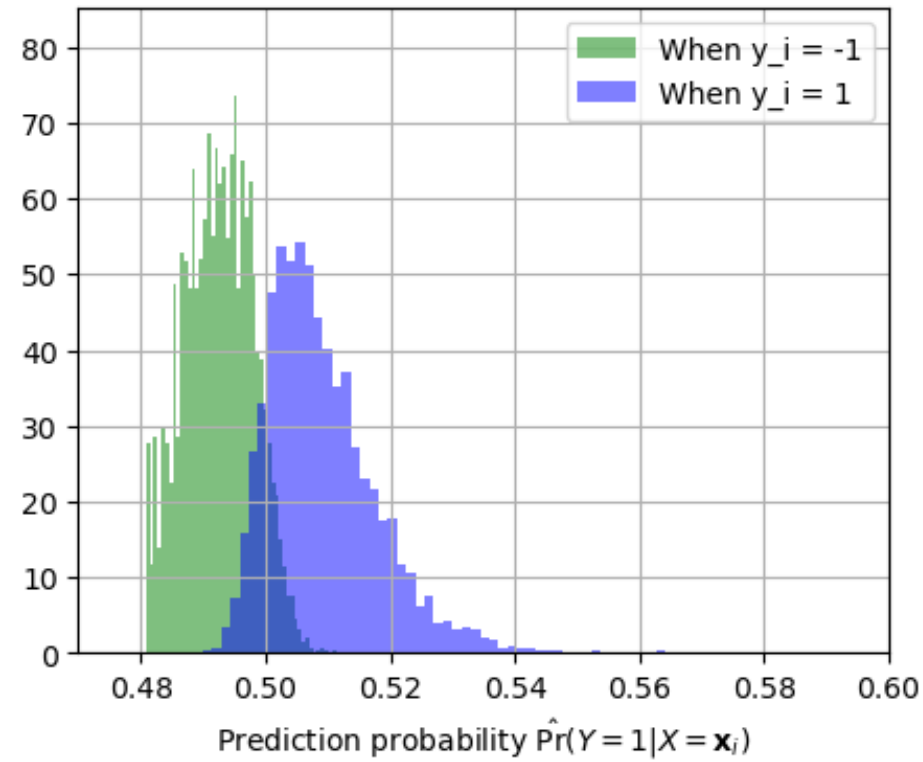
  where

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m G_m(\mathbf{x}) / \sum_{m=1}^{M} \alpha_m$$

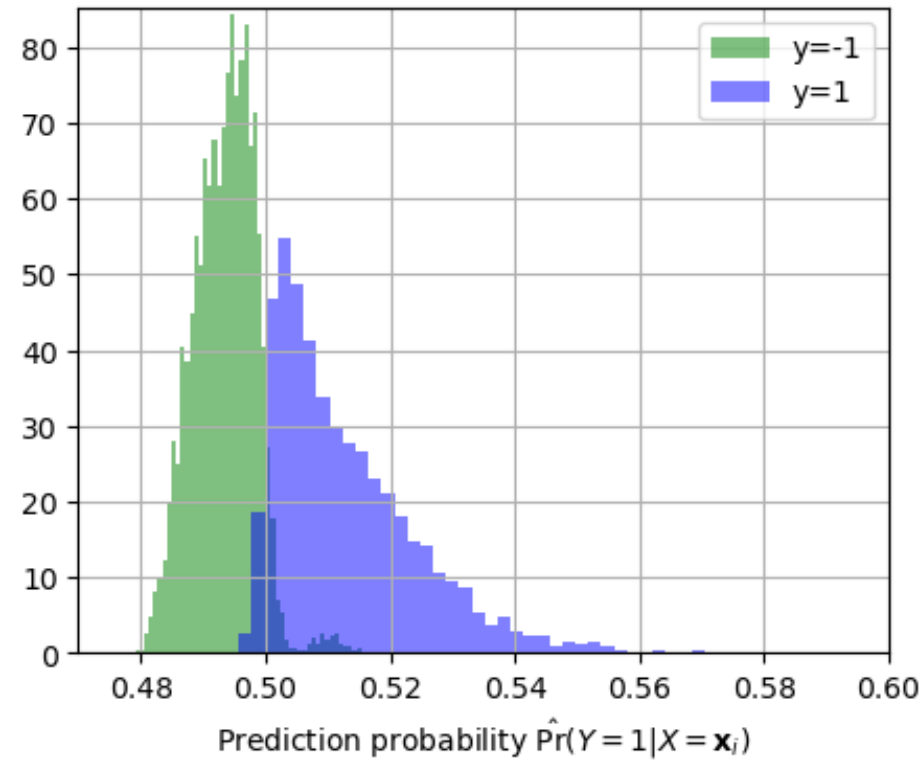
  is weighted mean predictions in the ensemble.
- This is what `.predict_proba` returns for `AdaBoostClassifier` with option `algorithm='SAMME'`.

Class prediction probabilities (continued)

Density histograms of prediction probabilities $\hat{p}(\mathbf{x})$ of test data for nested-spheres data set of Example 4.1.



AdaBoost.M1



AdaBoost.R

Discussion

We only touched a surface of vast literature on boosting in this chapter. Other popular methods:

- LogitBoost [Friedman et al., 2000] fits an additive logistic regression models by stagewise optimization of the Bernoulli log-likelihood. Instead of minimizing the exponential loss as AdaBoost (which is an approximation of the Bernoulli log-likelihood), it minimizes the Bernoulli log-likelihood directly.
- Gradient Boosting Machine (GBM) [Friedman, 2001] is a more general statistical framework for boosting that is based on its connection to **functional gradient descent (FGD)**, allowing to interpret boosting as a method for function estimation.
- XGBoost [Chen and Guestrin, 2016] is highly popular optimized distributed gradient boosting algorithm for large-scale data that builds on GBM framework. In mid 2010-s it was in many winning ML architectures in kaggle competitions.

Discussion

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 785–794, 2016.

Yoav Freund and Robert E Schapire. Experiments with a new boosting algorithm. In *Icml*, volume 96, pages 148–156. Citeseer, 1996.

Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2 edition, 2009.