# Statistical Methods in Machine Learning

## Trees

March 25, 2019

# Tree-based methods

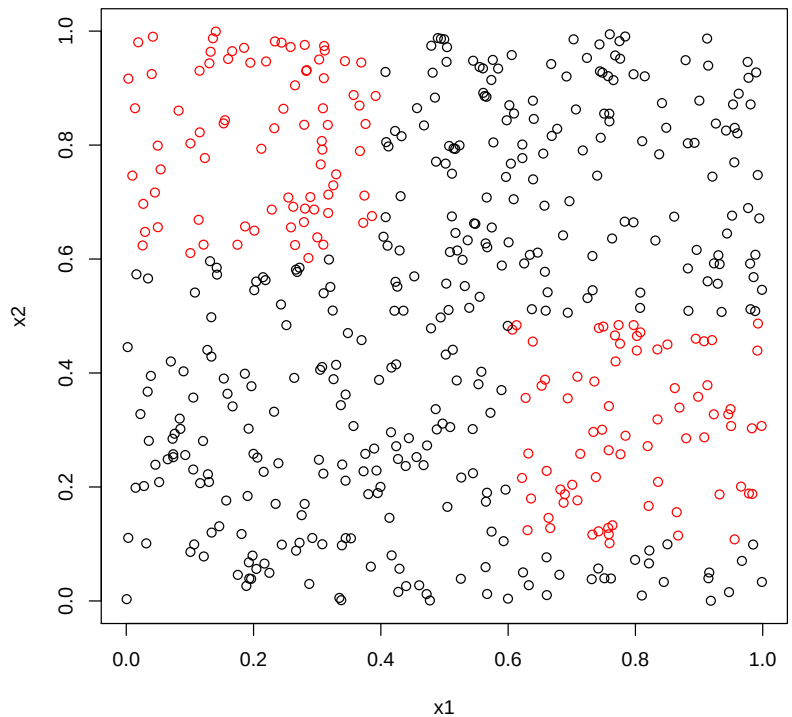
- Tree-based based methods for predicting $y$ from a feature vector $x \in \mathbb{R}^p$ divide up the feature space into rectangles, and then fit a very simple model in each rectangle. This works both when y is discrete and continuous, i.e., both for classification and regression
- Rectangles can be achieved by making successive binary splits on the predictors variables $X_1, \ldots, X_p$. I.e., we choose a variable $X_j, j = 1, \ldots, p$, divide up the feature space according to

$$X_j \leq c \text{ and } X_j > c$$

  Then we proceed on each half
- For simplicity, consider classification first (regression later). If a half is "pure", meaning that it mostly contains points from one class, then we don't need to continue splitting; otherwise, we continue splitting

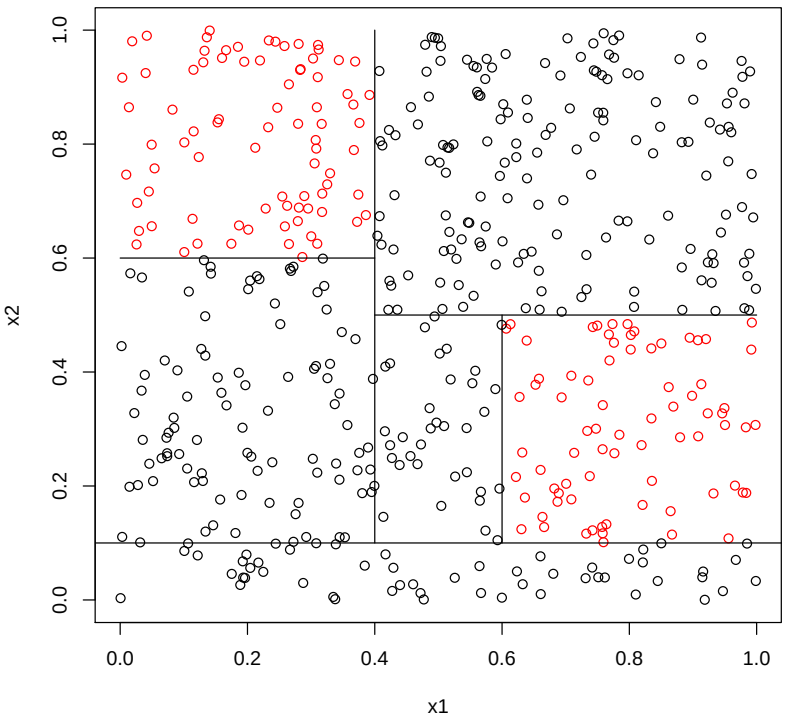
# Example: simple classification tree

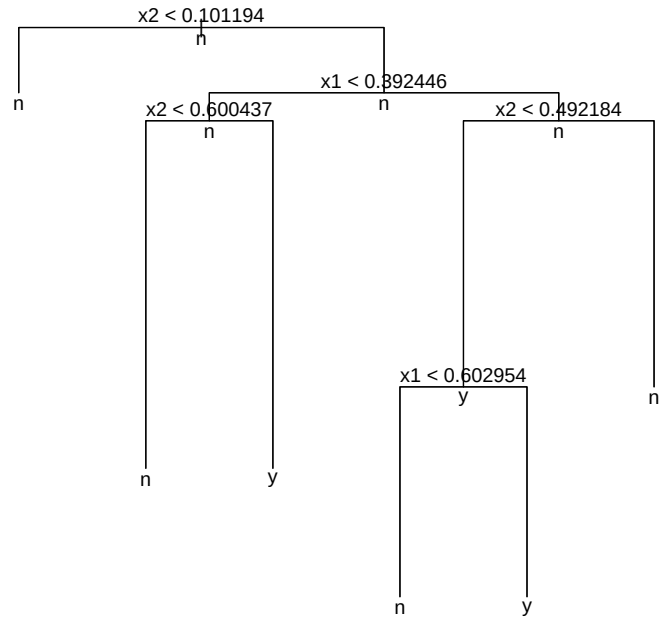
Example: $n = 500$ points in $p = 2$ dimensions, falling into classes 0 and 1, as marked by colors




- Could LDA or logistic regression do the job well?
- Does dividing up the feature space into rectangles look like it would work here?

# Example: simple classification tree (cont.)



# Terminology for Trees

- In keeping with the tree analogy, the rectangular are known as terminal nodes
- Decision trees are typically drawn upside down, in the sense that the leaves are at the bottom of the tree.
- The points along the tree where the predictor space is split are referred to as internal nodes

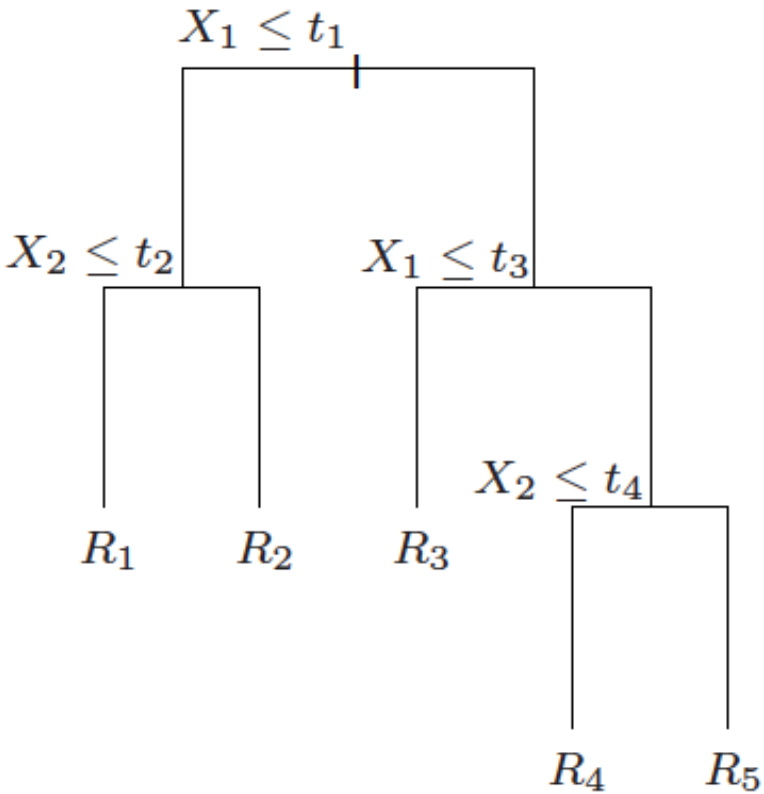
# Classification trees

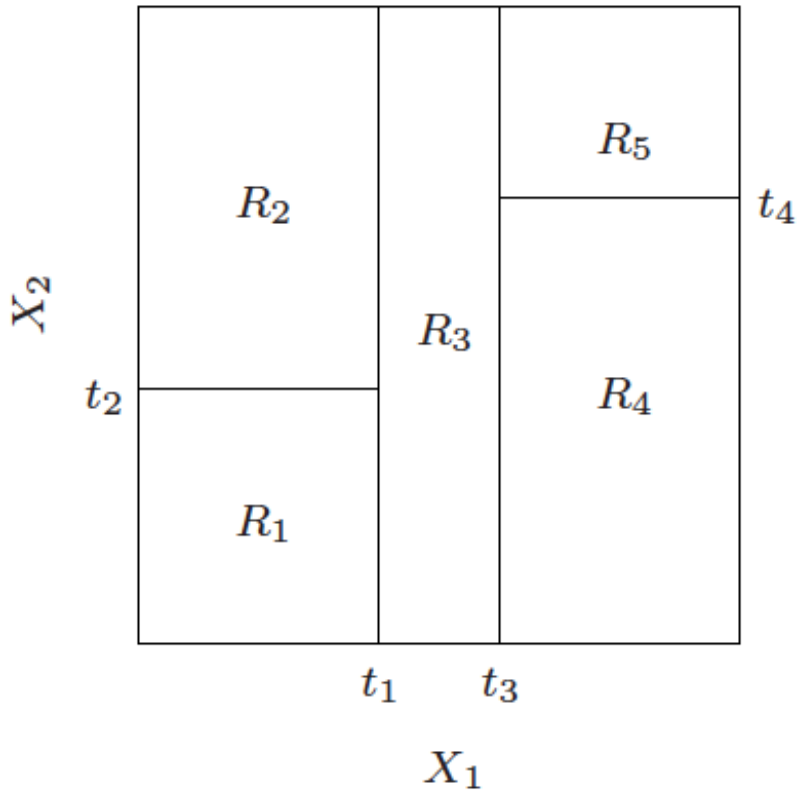
- Classification trees are popular because they are interpretable, and maybe also because they mimic the way (some) decisions are made
- Let $(x_i, y_i)$, $i = 1, \ldots, n$ be the training data, where $y_i \in \{1, \ldots, K\}$ are the class labels, and $x_i \in \mathbb{R}^p$ measure the p predictor variables. The classification tree can be thought of as defining m regions (rectangles) $R_1, \ldots, R_m$, each corresponding to a leaf of the tree
- We assign each $R_j$ a class label $c_j \in \{1, \ldots, K\}$. We then classify a new point $x$ by

$$\hat{f}^{\text{tree}}(x) = \sum_{j=1}^{m} c_j I(x \in R_j) = c_j, \text{ if } x \in R_j$$

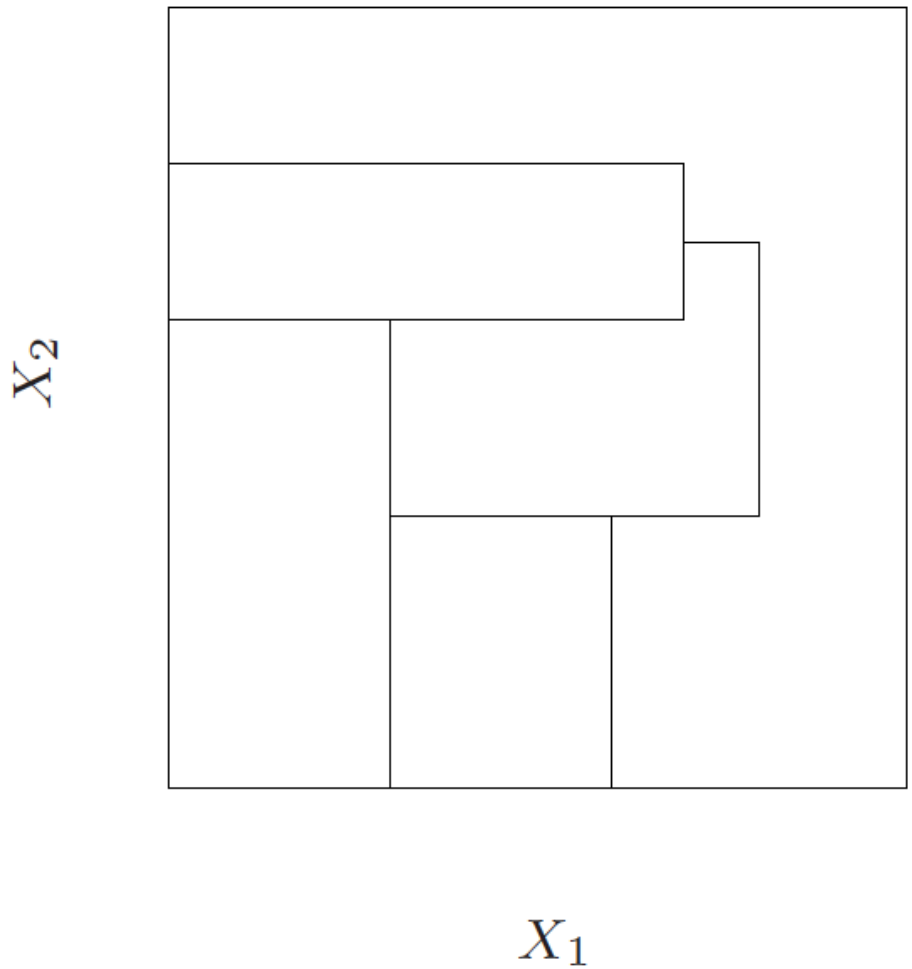
- Finding out which region a given point $x$ belongs to is easy since the regions $R_j$ are defined by a tree — we just scan down the tree. Otherwise, it would be a lot harder (need to look at each region)

# Example: regions defined by a tree



# Example: regions not defined by a tree



# Predicted class probabilities

- With classification trees, we can also get not only the predicted classes for new points but also the predicted class probabilties
- Note that each region $R_j$ contains some subset of the training data $(x_i, y_i)$ $i = 1, \ldots, n$, say, $n_j$ points. The predicted class $c_j$ is just most common occurring class among these points.
- Further, for each class $k = 1, \ldots, K$, we can estimate the probability that the class label is $k$ given that the feature vector lies in region $R_j$, $P(C = k | X \in R_j)$, by

$$\hat{p}_k(R_j) = \frac{1}{n_j} \sum_{x_i \in R_j} I(y_i = k)$$

  which is the proportion of points in the region that are of class $k$.
- We can now express the predicted class as

$$c_j = \arg\max_k \hat{p}_k(R_j)$$

# Trees provide a good balance

| | Model assumptions? | Estimated probabities? | Interpretable? | Flexible? | Predict well? |
|---|---|---|---|---|---|
| LDA | Yes | Yes | Yes | No | depends on $X$ |
| LR | Yes | Yes | Yes | No | depends on $X$ |
| k-NN | No | No | No | Yes | if properly tuned |
| Trees | No | Yes | Yes | Somewhat | ? |

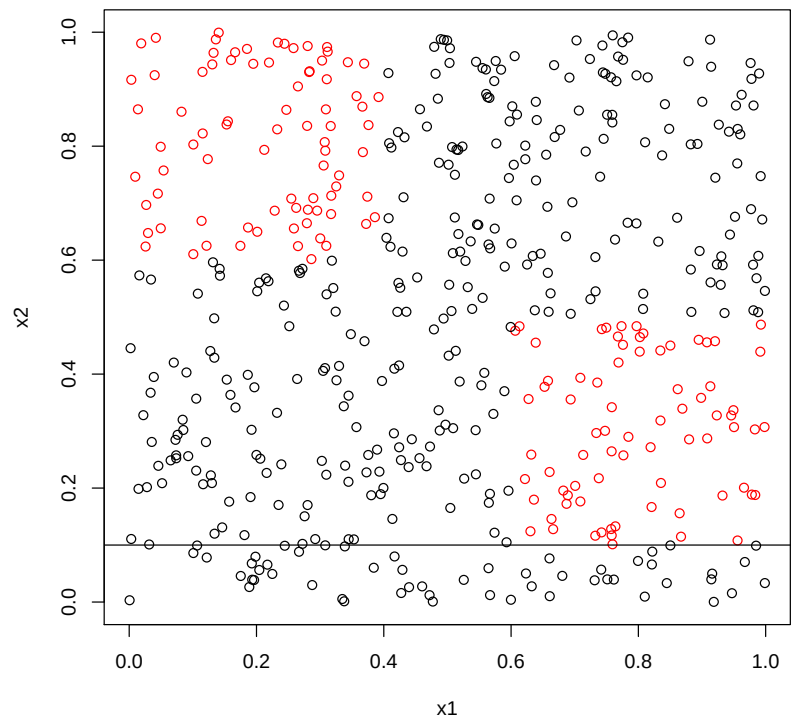
# How to build trees?

- There are two main issues to consider in building a tree:
  - How to choose the splits?
  - How big to grow the tree?
- Think first about varying the depth of the tree. Which is more complex, a big tree or a small tree? What tradeoff is at play here? How might we eventually consider choosing the depth?
- Now for a fixed depth, consider choosing the splits. If the tree has depth $d$ (and is balanced), then it has about $2^d$ nodes. At each node we could choose any of $p$ the variables for the split — this means that the number of possibilities is $p \times 2^d$
- This is huge even for moderate $d$! And we haven't even counted the actual split points themselves

# The CART algorithm

- The CART algorithm chooses the splits in a top down fashion: then chooses the first variable to at the root, then the variables at the second level, etc.
- At each stage we choose the split to achieve the biggest drop in misclassification error — this is called a greedy strategy. In terms of tree depth, the strategy is to grow a large tree and then prune at the end
- 

Why grow a large tree and prune, instead of just stopping at some point? Because any stopping rule may be short-sighted, in that a split may look bad but it may lead to a good split below it



# CART (cont.)

- Recall that in a region $R_m$, the proportion of points in class $k$ is

$$\hat{p}_k(R_m) = \frac{1}{n_m} \sum_{x_i \in R_m} I(y_i = k)$$

- The CART algorithm begins by considering splitting on variable $j$ and split point $s$, and defines the regions

$$R_1 = \{X \in \mathbb{R}^p; X_j \leq s\}, R_2 = \{X \in \mathbb{R}^p; X_j > s\}$$

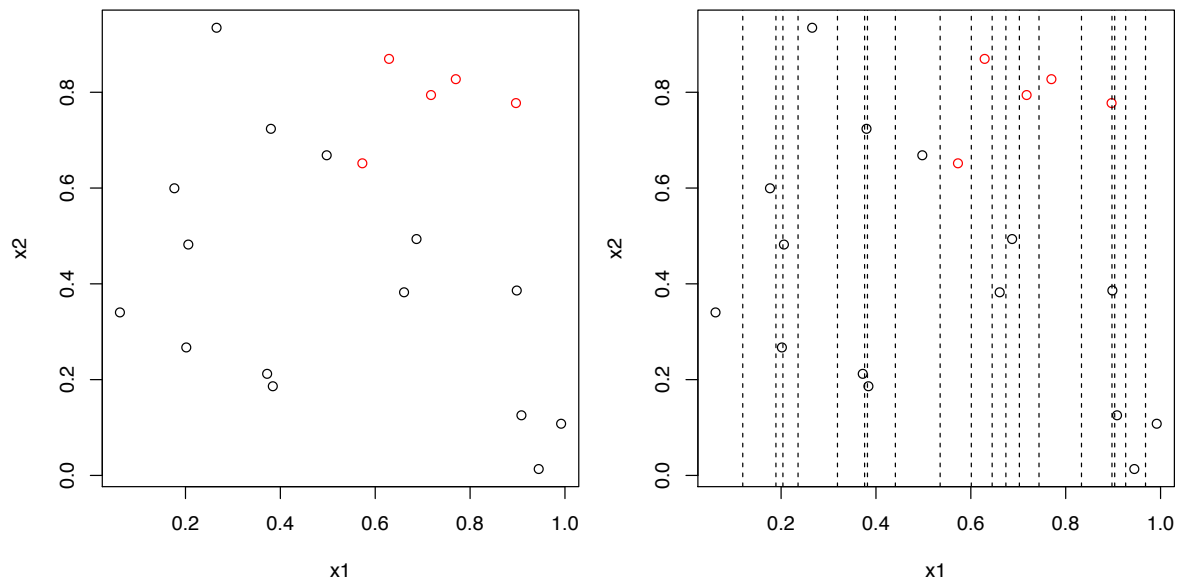
- We then greedily chooses j, s by minimizing the misclassification error

$$\arg\min_{j,s} \left( [1 - \hat{p}_{c_1}(R_1)] + [1 - \hat{p}_{c_2}(R_2)] \right)$$

where $c_1$ is the most common class in $R_1$, $c_2$ is the most common class in $R_2$

# CART (cont.)

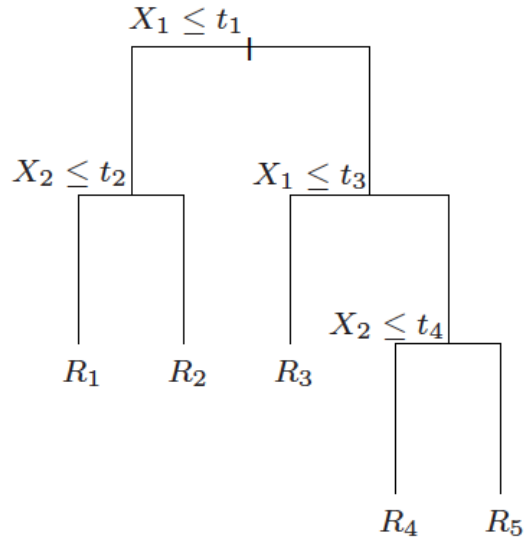
- Having done this, we now repeat this within each of the newly defined regions $R_1$, $R_2$.
- That is, it again considers splitting all variables $j$ and split points $s$, within each of $R_1$, $R_2$, this time greedily choosing the pair that provides us with the biggest improvement in misclassification error
- How do we find the best split s? Aren't there infinitely many to consider? No, to split a region $R_m$ on a variable $j$, we really only need to consider $n_m$ splits



# CART (cont.)

- 

Continuing on in this manner, we will get a big tree $T_0$. Its leaves define regions $R_1, \ldots, R_m$. We then prune this tree, meaning that we collapse some of its leaves into the parent nodes




- For any tree $T$, let $|T|$ denote its number of leaves. We define the cost complexity as

$$C_\alpha(T) = \sum_{j=1}^{|T|} [1 - \hat{p}_{c_j}(R_j)] + \alpha|T|$$

- We seek the tree $T \subset T_0$ that minimizes $C_\alpha(T)$. It turns out that this can be done by pruning the weakest leaf one at a time. Note that $\alpha$ is a tuning parameter, and a larger $\alpha$ yields a smaller tree. CART picks $\alpha$ by 5- or 10-fold cross-validation.

# Example

- To use CART in R, you can use either of the functions `rpart` or tree, in the packages of those same names.
- When you call `rpart`, cross-validation is performed automatically; when you call tree, you must then call `cv.tree` for cross-validation

- Example: $n = 4601$ emails, of which 1813 are considered spam. For each email we have $p = 58$ attributes. The first 54 measure the frequencies of 54 key words or characters (e.g., "free", "need", "$"). The last 3 measure
  - the average length of uninterrupted sequences of capitals;
  - the length of the longest uninterrupted sequence of capitals;
  - the sum of lengths of uninterrupted sequences of capitals

# Other impurity measures

- We used misclassification error as a measure of the impurity of region $R_j$,

$$1 - \hat{p}_{cj}(R_j)$$

- But there are other useful measures too: the Gini index:

$$\sum_{k=1}^{K} \hat{p}_{cj}(R_j)[1 - \hat{p}_{cj}(R_j)]$$

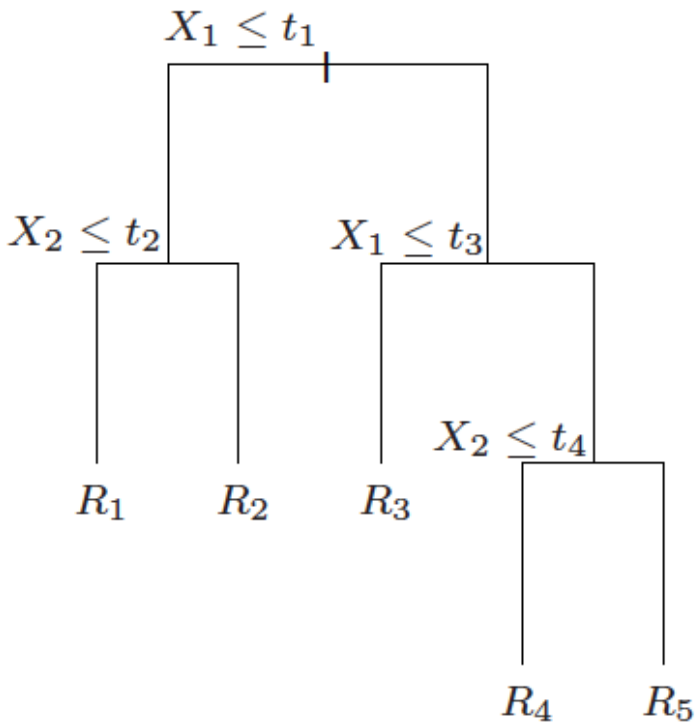
- and the cross-entropy or deviance:

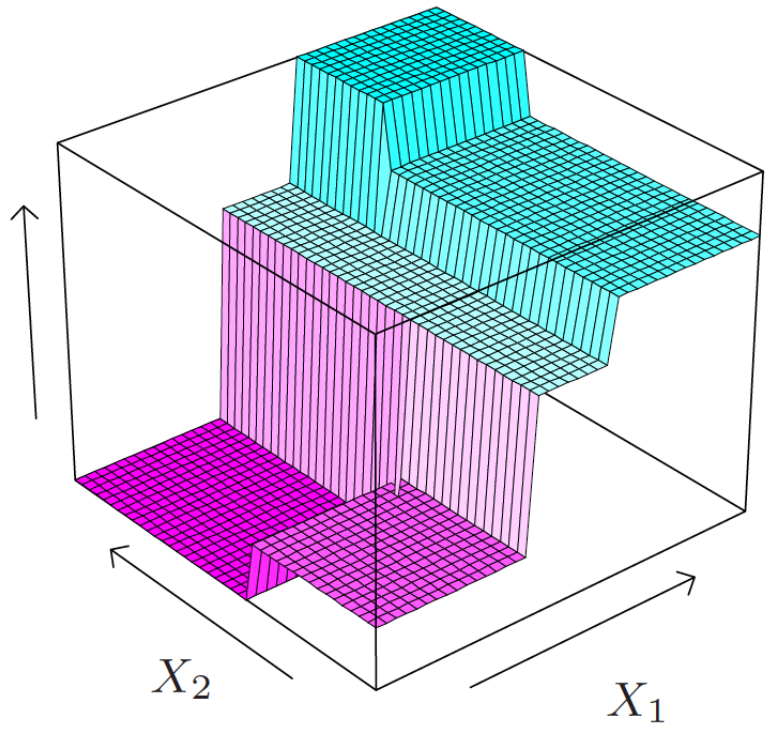
$$-\sum_{k=1}^{K} \hat{p}_{cj}(R_j) \log \hat{p}_{cj}(R_j)$$

- Using these measures instead of misclassification error is sometimes preferable because they are more sensitive to changes in class probabilities. Overall, they are all pretty similar.

# Regression trees

- Suppose that now we want to predict a continuous outcome instead of a class label. Essentially, everything follows as before, but now we just fit a constant inside each rectangle



# Regression Tree

- The estimated regression function has the form

$$\hat{f}^{\text{tree}}(x) = \sum_{j=1}^{m} c_j I(x \in R_j) = c_j, \text{ if } x \in R_j$$

- just as it did with classification. The quantities $c_j$ are no longer predicted classes, but instead they are real numbers. How would we choose these?
- Simple: just take the average response of all of the points in the region,

$$c_j = \frac{1}{n_j} \sum_{x_i \in R_j} y_i$$

- The main difference in building the tree is that we use squared error loss instead of misclassification error to decide which region to split. Also, with squared error loss, choosing $c_j$ as above is optimal

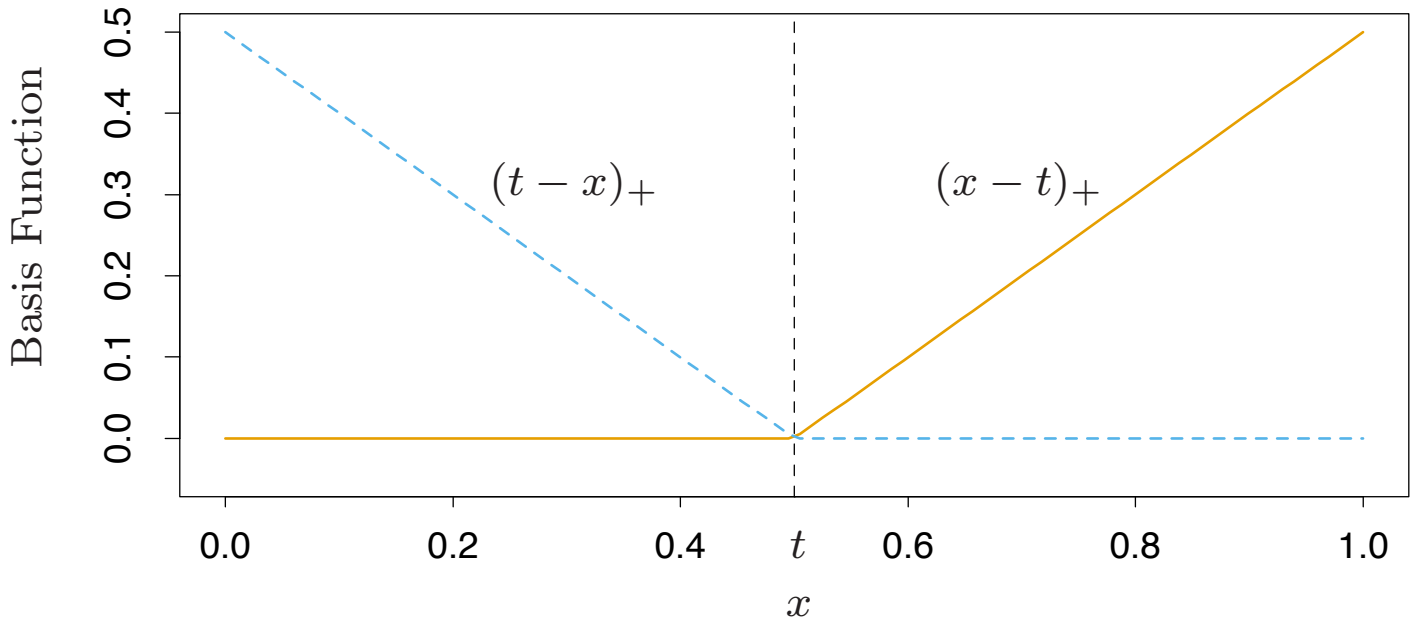
# How well do trees predict?

- Trees seem to have a lot of things going in the favor. So how is their predictive ability?
- Unfortunately, the answer is not great.
- Of course, at a high level, the prediction error is governed by bias and variance, which in turn have some relationship with the size of the tree (number of nodes).
- A larger size means smaller bias and higher variance, and a smaller tree means larger bias and smaller variance
- But trees generally suffer from high variance because they are quite instable: a smaller change in the observed data can lead to a dramatically different sequence of splits, and hence a different prediction.
- This instability comes from their hierarchical nature; once a split is made, it is permanent and can never be "unmake" further down in the tree
- We'll learn some variations of trees have much better predictive abilities. However, their predictions rules aren't as transparent

# Extension: Multivariate Adaptive Regression Splines

- Multivariate Adaptive Regression Splines (MARS) is an adaptive procedure for regression,
- It can be viewed as a a modification of the CART method to improve the performance in the regression setting.
- MARS uses expansions in piecewise linear basis functions of the form

$$(x-t)_+ \text{ and } (t-x)_+$$



# Model representation

- Each function is piecewise linear, with a knot at the value $t$. In the terminology of previous lectures, these are linear splines. We call the two functions a reflected pair
- Suppose we have predictor $(X_1, \ldots, X_p) \in \mathbb{R}^p$, the collection of basis functions is

$$\mathcal{C} = \{(X_j - t)_+, (t - X_j)_+\}, \quad t \in \{x_1, \ldots, x_n\}, \ j = 1, .., p$$

- If all of the input values are distinct, there are $2Np$ basis functions altogether.
- We are allowed to use functions from the set $\mathcal{C}$ and their products

$$f(X) = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(X)$$

where each $h_m(X)$ is a function in $C$, or a product of two or more such functions.

# Model building

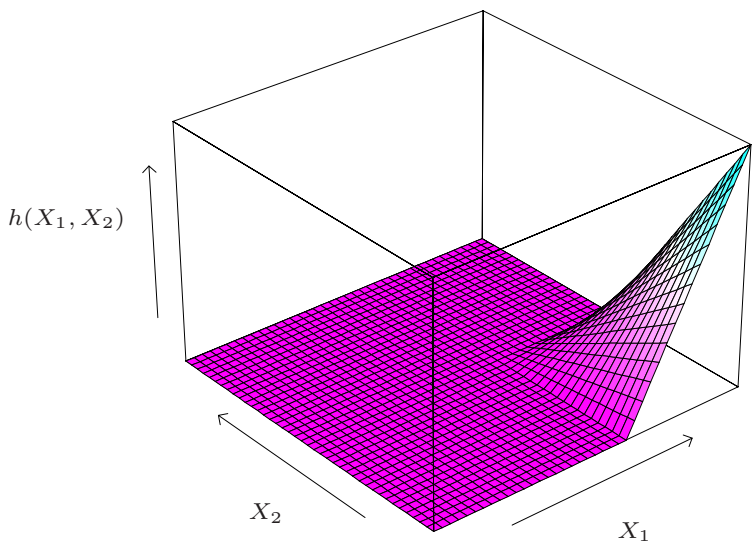
- At each stage we have a model

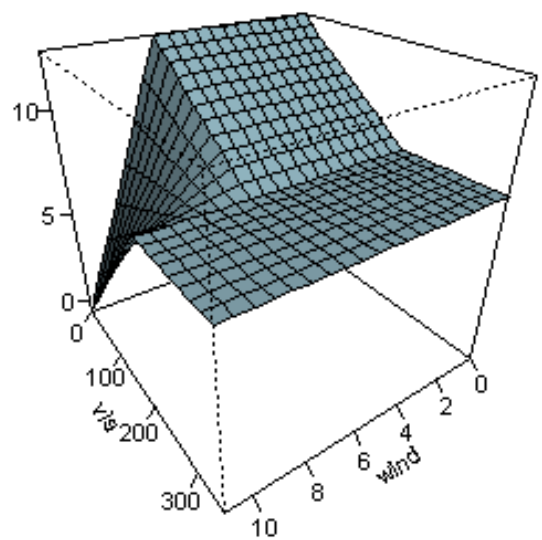
$$f_M(X) = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(X)$$

- At each stage we consider as a new basis function pair all products of a function $h_m$ in the model set $\mathcal{M}$ with one of the reflected pairs in $C$. We add to the model $\mathcal{M}$ the term of the form

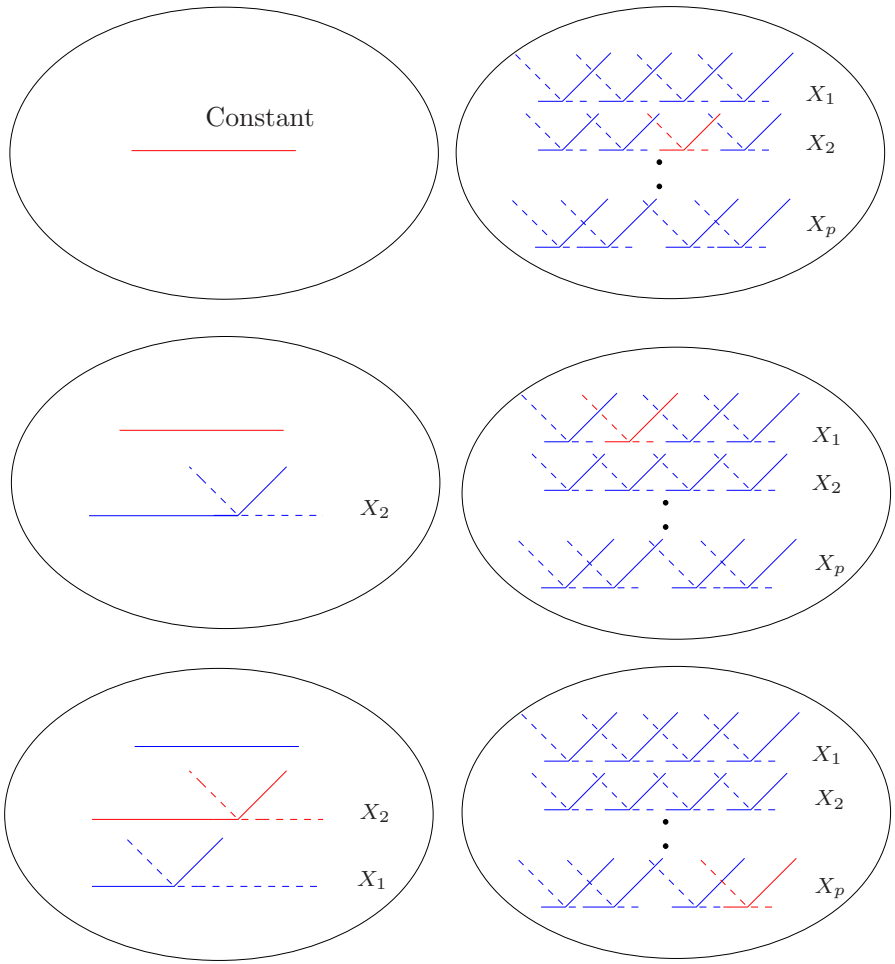
$$\beta_{M+1} h_l(X) \times (X_j - t)_+ + \beta_{M+2} h_l(X) \times (t - X_j)_+, \quad h_l \in \mathcal{M}$$

- Interaction



# Model Building (cont.)




In R, you can use the `earth` package.

# Relationship of MARS to CART

- Although they might seem quite different, the MARS and CART strategies actually have strong similarities.
- Suppose we take the MARS procedure and make the following changes:
  - Replace the piecewise linear basis functions by step functions
  - When a model term is involved in a multiplication by a candidate term, it gets replaced by the interaction
- With these changes, the MARS forward procedure is the same as the CART tree-growing algorithm.
- The second restriction implies that a node may not be split more than once, and leads to the attractive binary-tree representation of the CART model.
- On the other hand, it is this restriction that makes it difficult for CART to model additive structures. MARS forgoes the tree structure and gains the ability to capture additive effects.

# Bagging

- **B**ootstrap **agg**regat**ing** = Bagging
- Proposed by Leo Breiman
  `http://www.stat.berkeley.edu/~breiman/`
- Wikipedia:
  *Bootstrap aggregating (bagging) is a* **machine learning ensemble** *meta-algorithm to improve machine learning of statistical classification and regression models in terms of* **stability** *and classification* **accuracy**. *It also reduces* **variance** *and helps to avoid* **overfitting**.
- Although it is usually applied to decision tree models, it can be used with any type of model.

# Idea

- Bootstrap aggregation, or bagging, is a general-purpose procedure for reducing the variance of a statistical learning method
- Recall that given a set of $n$ independent observations $Z_1, \ldots, Z_n$, each with variance $\sigma^2$, the variance of the mean $\bar{Z}$ is given by $\sigma^2/n$
- In other words, averaging a set of observations reduces variance. Of course, this is not practical because we generally do not have access to multiple training sets.

# What is Bagging?

- Training set $T = \{(x_i, y_i)\}$
- given $x$, we have predictor $\varphi(x, T)$
- Now, we are given a seq of training set $\{T_k\}$
- Denote $\varphi_A(x)$ be the averages of $\varphi(x, T_k)$
  - if $y$ is numerical,
$$\varphi_A(x) = E_T \varphi(x, T)$$
  - if $y$ is class label, we compute $\varphi_A(x)$ by voting (more later)
- Usually, we only have one training set. We can generate $T_k$ by bootstrapping. The resulting predictor is $\varphi_B(x)$ and this procedure is called **Bagging**.

# Recall: Bootstrap

- A bootstrap sample of size $m$ from the training data is

$$(x_i^*, y_i^*), i = 1, \ldots, m$$

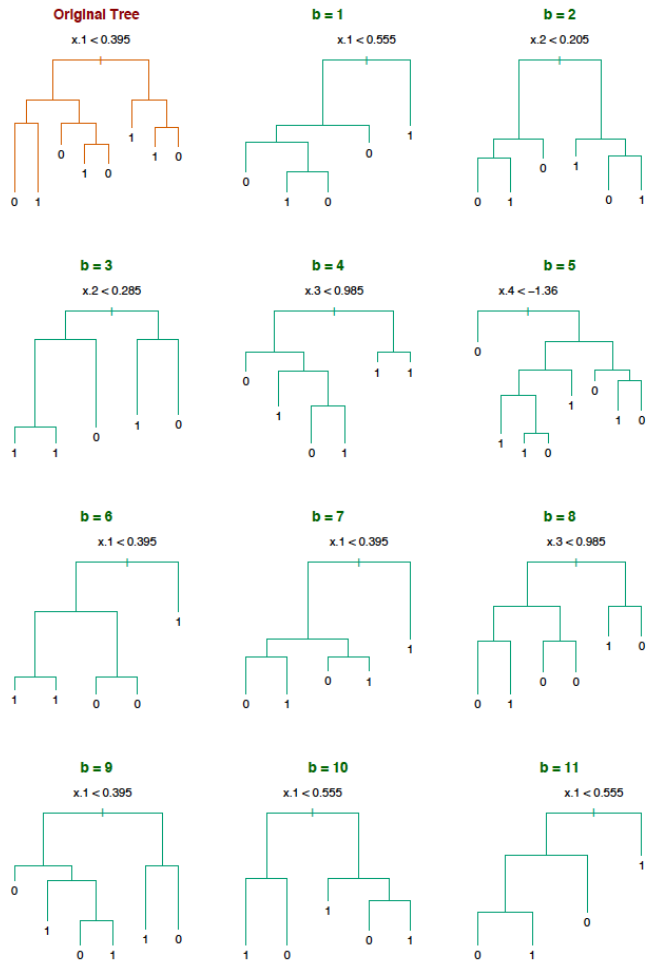
  where each $(x_i^*, y_i^*)$ are drawn from uniformly at random from $(x_i, y_i)_{i=1}^n$, with replacement
- This corresponds exactly to m independent draws from the empirical distribution $\hat{F}$. Hence it approximates what we would see if we could sample more data from the true $F$. We often consider $m = n$, which is like sampling an entirely new training set
- Note: not all of the training points are represented in a bootstrap sample, and some are represented more than once. When $m = n$, about $1/3$ of points are left out, for large $n$.

# Bagging in Classification Tree

- Given a training data $(x_i, y_i), i = 1, \ldots, n$, bagging averages the predictions from classification trees over a collection of boostrap samples.
- For $b = 1, \ldots, B$ (e.g., $B = 100$), we draw n boostrap samples $(x_i^{*b}, y_i^{*b}), i = 1, \ldots, n$, and we fit a classification tree $\hat{f}^{\text{tree},b}$ on this sampled data set. Then at the end, to classify an input $x$ we simply take the most commonly predicted class:
- $$\hat{f}^{\text{bag}}(x) = \underset{k=1,\ldots,K}{\arg\max} \sum_{b=1}^{B} I(\hat{f}^{\text{tree},b}(x) = k)$$
- Two options to build trees:
  - Simple strategy: grow fairly large trees on each sampled data set, with no pruning
  - More involved strategy: prune back each tree as we do with CART, but use the original training data as the validation set, instead of performing cross-validation

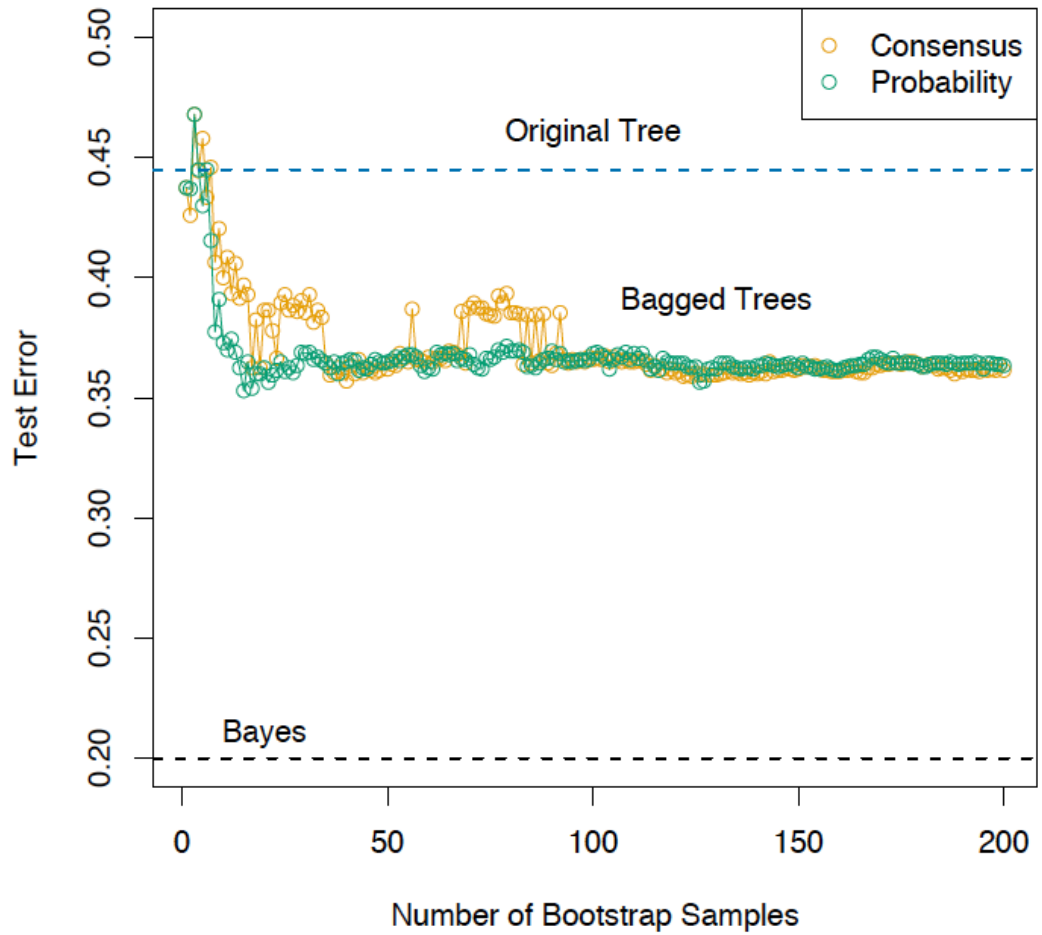
# Example: bagging

There are $n = 30$ training data points, $p = 5$ features, and $K = 2$ classes. No pruning used in growing trees:



# Example: bagging (cont.)

Bagging helps decrease the misclassification rate of the classifier (evaluated on a large independent test set).



# Voting probabilities are not estimated class probabilities

- Suppose that we wanted estimated class probabilities out of our bagging procedure. What about using, for each $k = 1, \ldots, K$

$$\hat{p}_k^{\text{bag}}(x) = \arg\max_{k=1,\ldots,K} \frac{1}{B} \sum_{b=1}^{B} I(\hat{f}^{\text{tree},b} = k)$$

- i.e., the proportion of votes that were for class $k$.
- This is generally not a good estimate. Simple example: suppose that the true probability of class 1 given $x$ is 0.75.
- Suppose also that each of the bagged classifiers $\hat{f}^{\text{tree},b}$ correctly predicts the bag class to be 1. Then $\hat{f}^{\text{tree},b}$, which is wrong

# Alternative form of bagging

- Each tree already gives us a set of predicted class probabilities at $x$: $\hat{p}_k^{\text{tree},b}$
- This suggests an alternative method for bagging. Instead of simply taking the prediction $\hat{f}^{\text{tree},b}$ from each tree, we go further and look at its predicted class probabilities

$$\hat{p}_k^{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{p}_k^{\text{tree},b}(x)$$

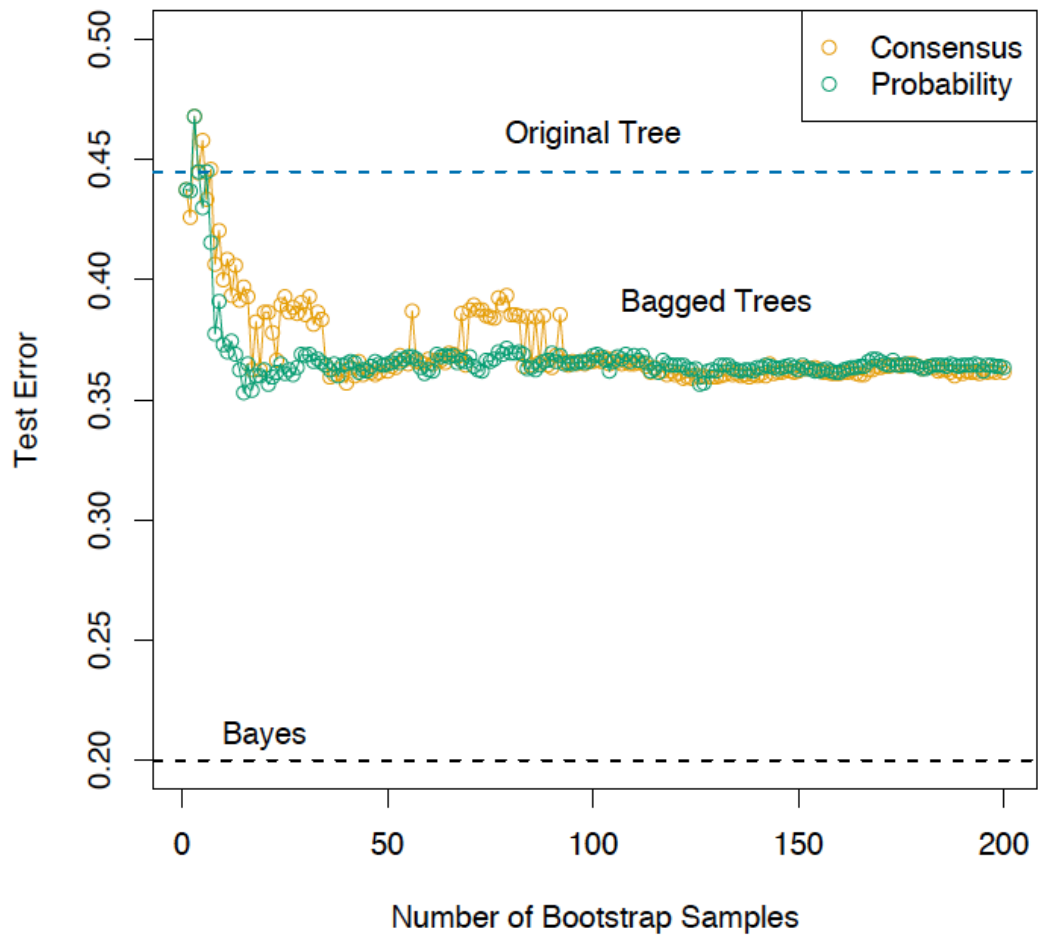
- The final bagged classifier just chooses the class with the highest probability:

$$\hat{f}^{\text{bag}}(x) = \underset{k=1,\ldots,K}{\arg\max}\, \hat{p}_k^{\text{bag}}(x)$$

- This form of bagging is preferred if it is desired to get estimates of the class probabilities. Also, it can sometimes help the overall prediction accuracy

# Example: alternative form of bagging

Previous example revisited: the alternative form of bagging produces misclassification errors shown in green



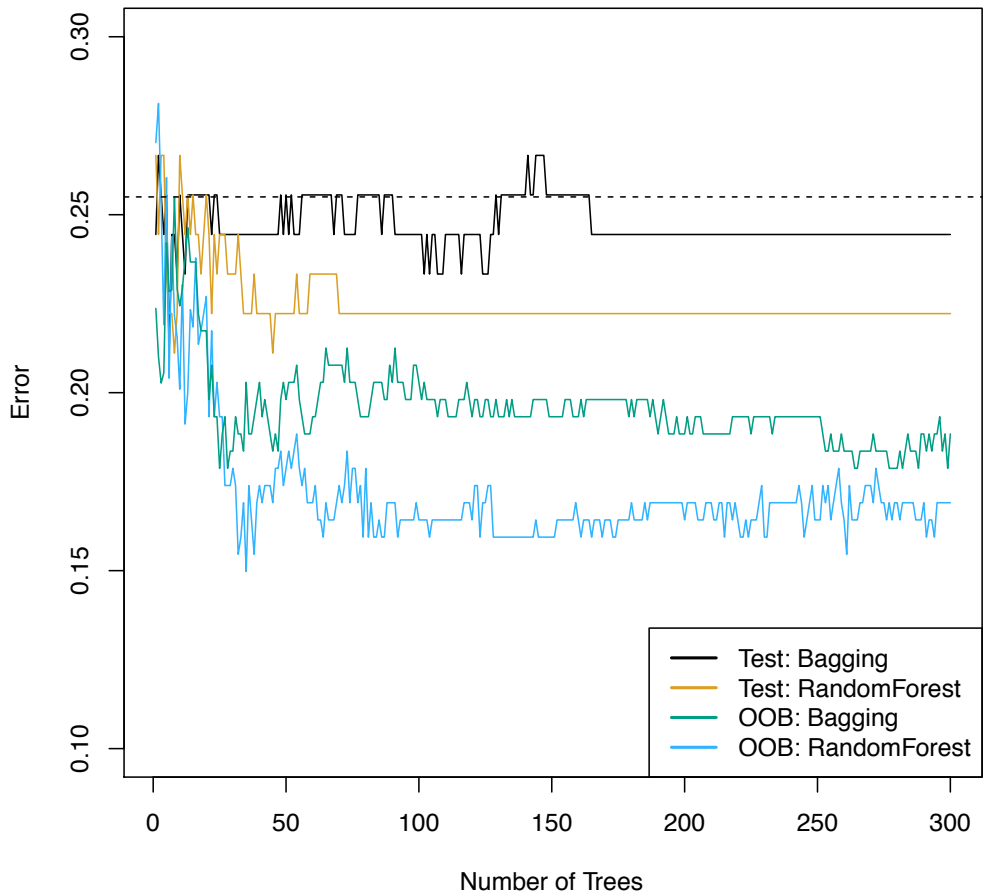
# Out-of-Bag Error Estimation

- It turns out that there is a very straightforward way to estimate the test error of a bagged model.
- Recall that the key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations. One can show that on average, each bagged tree makes use of around two-thirds of the observations.
- The remaining one-third of the observations not used to fit a given bagged tree are referred to as the out-of-bag (OOB) observations.
- We can predict the response for the $i$th observation using each of the trees in which that observation was OOB. This will yield around $B/3$ predictions for the $i$th observation, which we average.
- This estimate is essentially the LOO cross-validation error for bagging, if $B$ is large.
- Unlike cross-validation, these require no additional computing

# Random Forests

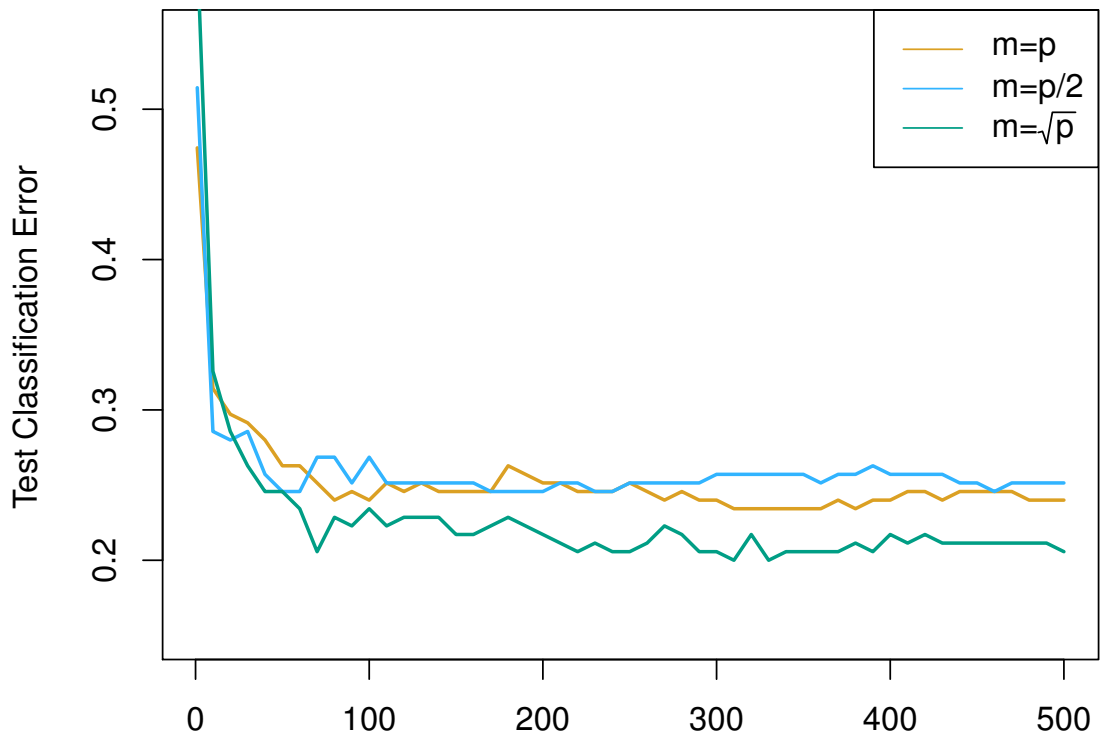
- Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees. This reduces the variance when we average the trees.
- As in bagging, we build a number of decision trees on bootstrapped training samples.
- But when building these decision trees, each time a split in a tree is considered, a random selection of $m$ predictors is chosen as split candidates from the full set of $p$ predictors. The split is allowed to use only one of those $m$ predictor
- A fresh selection of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$

# Example


0.30
0.25
0.20
0.15
0.10
Error
0
50
100
150
200
250
300
Number of Trees
Test: Bagging
Test: RandomForest
OOB: Bagging
OOB: RandomForest

# Why $\sqrt{p}$?

- Results from random forests for the fifteen-class gene expression data set with $p = 500$predictors.
- The test error is displayed as a function of the number of trees. Each colored line corresponds to a different value of $m$, the number of predictors available for splitting at each interior tree node.
- Random forests ($m < p$) lead to a slight improvement over bagging ($m = p$). A single classification tree has an error rate of 45.7%.



# Variable importance measure

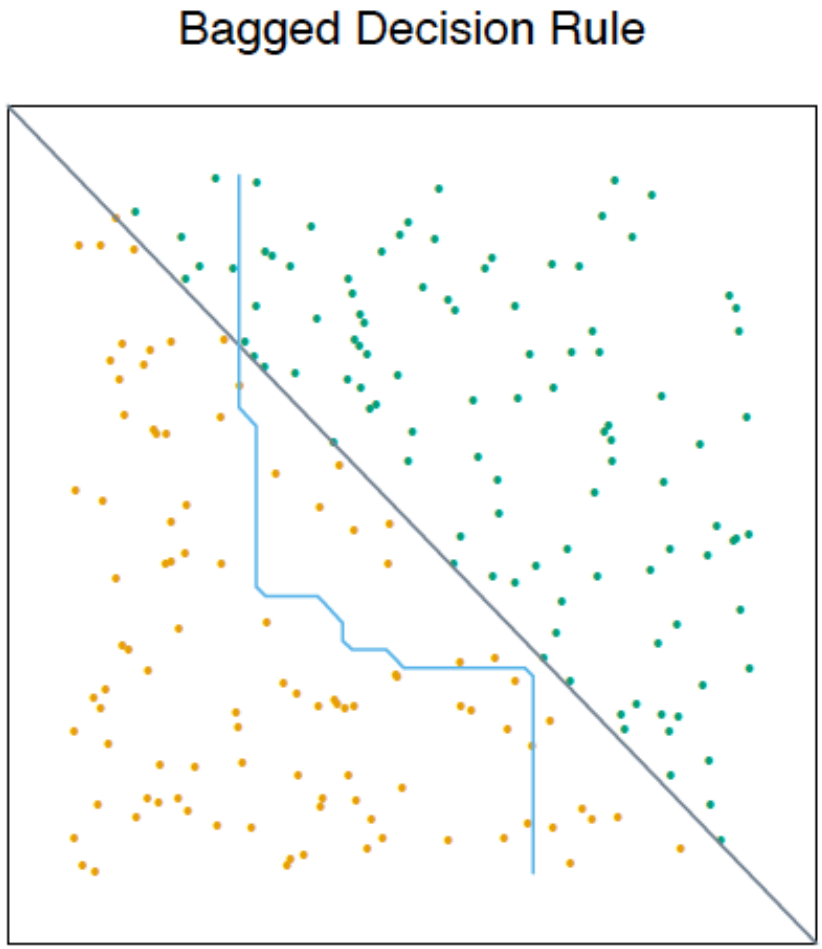
- Bagging typically results in improved accuracy over prediction using a single tree. Unfortunately, however, it can be difficult to interpret the resulting model
- However, one can obtain an overall summary of the importance of each predictor
- the total amount that the RSS/Gini index is decreased due to splits over a given predictor, averaged over all $B$ trees. A large value indicates an important predictor.
- more later when we talk about boosting

# Disadvantages

- Loss of interpretability: the final bagged classifier is not a tree, and so we forfeit the clear interpretative ability of a classification tree
- Computational complexity: we are essentially multiplying the work of growing a single tree by $B$ (especially if we are using the more involved implementation that prunes and validates on the original training data)
- You can think of bagging as extending the space of models. We go from fitting a single tree to a large group of trees. Note that the final prediction rule cannot always be represented by a single tree
- Sometimes, this enlargement of the model space isn't enough, and we would benefit from an even greater enlargement

# Example: limited model space

Bagging still can't really represent a diagonal decision rule



# Classification trees with observation weights

- Now suppose that we are are additionally given observation weights $w_i, i = 1, \ldots, n$. Each $w_i \geq 0$, and a higher value means that we place a higher importance in correctly classifying this observation
- The CART algorithm can be easily adapted to use the weights. All that changes is that our sums become weighted sums. I.e., we now use the weighted proportion of points of class $k$ in region $R_m$:

$$\hat{p}_k(R_m) = \frac{\sum_{x_i \in R_m} w_i I(y_k = k)}{\sum_{x_i \in R_m} w_i}$$

- As before, we let

$$c_m = \arg\max_k \hat{p}_k(R_m)$$

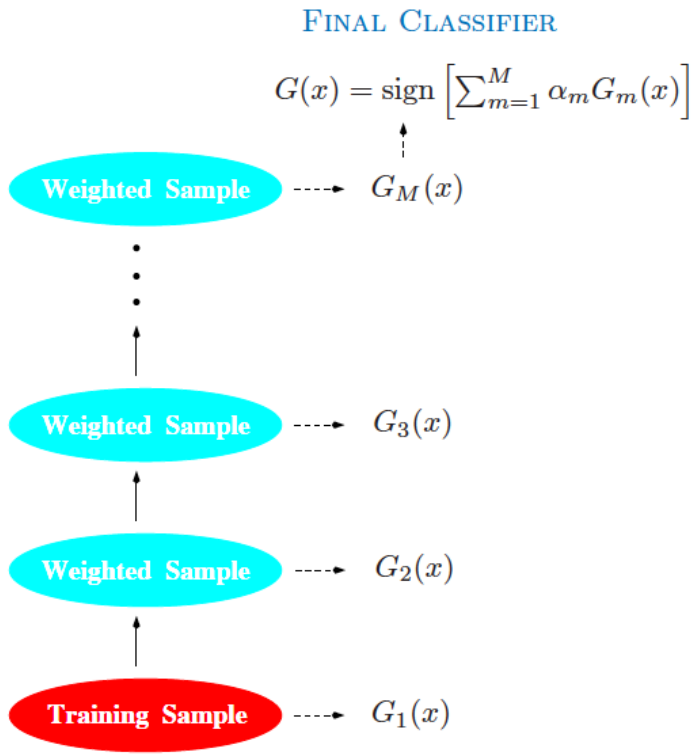
- and hence $1 - \hat{p}_{c_m}(R_m)$ is the weighted misclassification error

# Boosting

- Boosting is similar to bagging in that we combine the results of several classification trees. However, boosting does something fundamentally different, and can work a lot better
- As usual, we start with training data $(x_i, y_i), i = 1, \ldots, n$. We'll assume that $y_i \in \{-1, 1\}$ and $x_i \in \mathbb{R}^p$. Hence we suppose that a classification tree (fit, e.g., to the training data) will return a prediction of the form $\hat{f}^{\text{tree}}(x) = \in \{-1, 1\}$ for an input $x \in \mathbb{R}^p$
- In boosting we combine a weighted sum of $B$ different tree classifiers,

$$\hat{f}^{\text{boost}}(x) = \text{sign}\left(\sum_{b=1}^{B} \alpha_b \hat{f}^{\text{tree},b}(x)\right)$$

- One of the key differences between boosting and bagging is how the individual classifiers $\hat{f}^{\text{tree},b}(x)$ are fit.
- Unlike in bagging, in boosting we fit the tree to the entire training set, but adaptively weight the observations to encourage better predictions for points that were previously misclassified



# The basic boosting algorithm (AdaBoost)

- Given training data $(x_i, y_i), i = 1, \ldots, n$, the basic boosting method is called AdaBoost, and can be described as:
- Initialize the weights by $w_i = 1/n$ for each $i$
- For $b = 1, \ldots, B$:
  1. Fit a classification tree $\hat{f}^{\text{tree},b}(x)$ to the training data with weights $w_1, \ldots, w_n$
  2. Compute the weighted misclassification error

$$e_b = \frac{\sum_{i=1}^{n} w_i I(y_i \neq \hat{f}^{\text{tree},b}(x_i))}{\sum_{i=1}^{n} w_i}$$

  3. Let $\alpha_b = \log\{(1 - e_b)/e_b\}$
  4. Update the weights as

$$w_i \leftarrow w_i \times \exp\left[\alpha_b I(y_i \neq \hat{f}^{\text{tree},b}(x_i))\right]$$

- $\hat{f}^{\text{boost}} = \text{sign}\left(\sum_{b=1}^{B} \alpha_b \hat{f}^{\text{tree},b}(x)\right)$

# Example

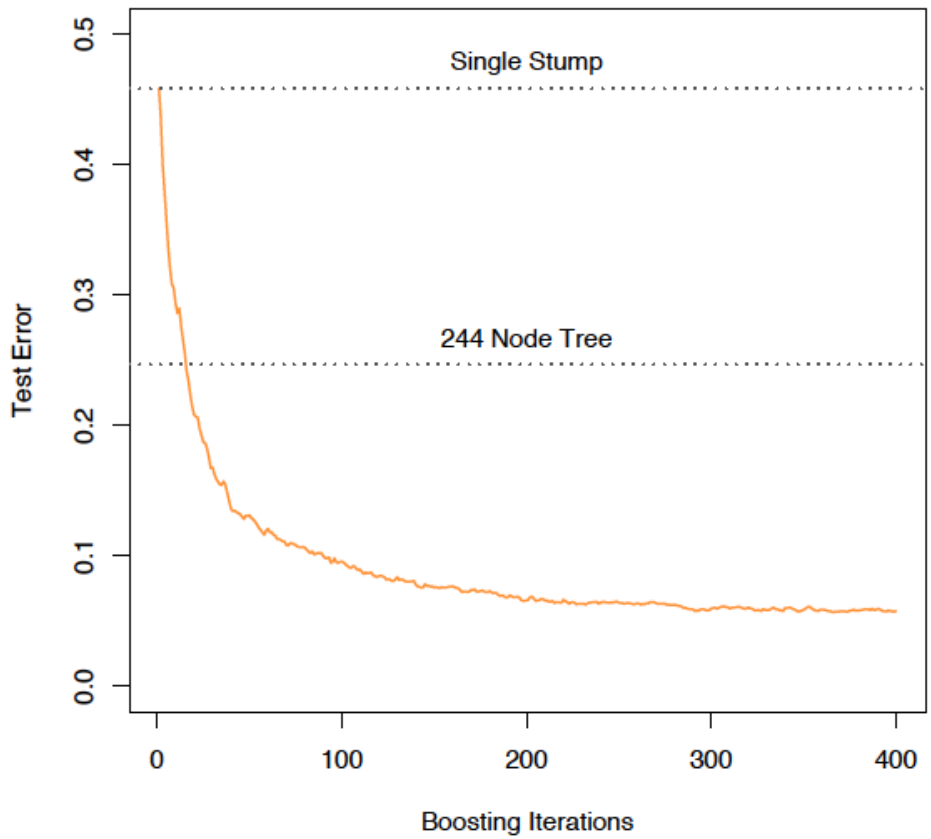
- Here $n = 1000$ points were drawn from the model

$$Y_i = \begin{cases} 1, & \text{if } \sum_{j=1}^{10} X_{ij}^2 > \chi_{10}^2(0.5) \\ -1, & \text{otherwise} \end{cases}$$

  where each $X_{ij} \sim N(0,1)$ independently, $j = 1, \ldots, 10$
- A stump classifier was computed: this is just a classification tree with one split (two leaves). This has a bad misclassification rate on independent test data, of about 45.8%.

# Example (cont.)

On the other hand, boosting stumps achieves an excellent error rate of about 5.8% after $B = 400$ iterations. This is much better than, e.g., a single large tree (error rate 24.7%)



# Why does boosting work?

- The intuition beyond boosting is simple: we weight misclassified observations in such a way that they get properly classified in future iterations.
- But there are many ways to do this — why does the particular weighting seem to work so well?
- One nice discovery was the connection between boosting and forward stepwise modeling
- To understand this connection, it helps to recall forward stepwise linear regression. Given a continuous response $y$ and predictors $X_1, \ldots, X_p$
  1. Choose the predictor $X_j$ giving the smallest squared error loss

$$\sum_{i=1}^{n} \left( y_i - \hat{\beta}_j X_{ij} \right)^2$$

  2. $r_i = y_i - \beta_j X_{ij}$
  3. Choose the predictor $X_k$ giving the smallest additional loss

$$\sum_{i=1}^{n} \left( r_i - \hat{\beta}_j X_{ik} \right)^2$$

  4. Repeat

# Boosting fits an additive model

- In the classification setting, $y_i$ is not continuous but discrete, taking values in $\{1, 1\}$. Therefore squared error loss is not appropriate. Instead we use exponential loss:

$$L(y_i, f(x_i)) = \exp(-y_i f(x_i))$$

- Furthermore, boosting does not model $f(x_i)$ as a linear function of the predictors, but rather as a linear function of trees
- Hence the analogous forward stepwise modeling is as follows:
  1. Choose the tree $T_j$ and coefficient $\beta_j$ giving the smallest exponential loss

$$\sum_{i=1}^{n} \exp(-y_i \beta_j T_j(x_i))$$

  2. Choose the tree $T_k$ and coefficient $\beta_k$ giving the smallest additional loss

$$\sum_{i=1}^{n} \exp\left[-y_i \left\{\beta_j T_j(x_i) + \beta_k T_k(x_i)\right\}\right]$$

$$= \sum_{i=1}^{n} \exp(-y_i \beta_j T_j(x_i)) \exp(-y_i \beta_k T_k(x_i))$$

  3. repeat

- This forward stepwise procedure can be tied concretely to the AdaBoost algorithm that we just learned
- This connection brought boosting from an unfamiliar algorithm to a more-or-less familiar statistical concept. Consequently, there were many suggested ways to extend boosting:
  - Other losses: exponential loss is not really used anywhere else. It leads to a computationally efficient boosting algorithm (AdaBoost), but binomial deviance and SVM loss are two alternatives that can work better (gradient boosting).
  - Shrinkage: instead of adding one tree at a time, why don't we add a small multiple of a tree? This is called shrinkage a form of regularization for the model building process. This can also be very helpful in terms of prediction accuracy

# Gradient Boost Algorithm for regression tree

1. Initiate $f_0(x) = \arg\min_\gamma \sum_{i=1}^n L(y_i, \gamma)$
2. For $m = 1, \ldots, M$
   (A) For $i = 1, \ldots, n$, compute

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

   (B) Fit a tree to the target $r_{im}$, $i = 1, \ldots, n$, denote the terminal regions as $R_{jm}$ where $j = 1, \ldots, J_m$
   (C) For $j = 1, \ldots, J_m$ compute

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

   (D) Set $f_m(x) = f_{m-1} + \lambda \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$. The parameter $\lambda$ here is called shrinkage parameter or learning rate.
3. $\hat{f}(x) = f_M(x)$

# Disadvantages

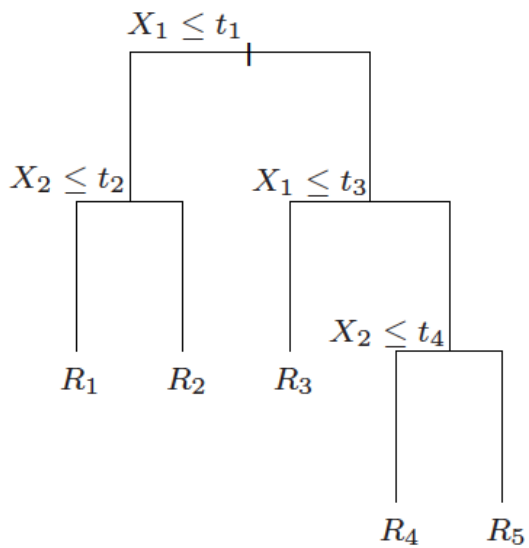
- As with bagging, a major downside is that we lose the simple interpretability of classification trees. The final classifier is a weighted sum of trees, which cannot necessarily be represented by a single tree.
- Computation is also somewhat more difficult. However, with AdaBoost, the computation is straightforward (more so than gradient boosting). Further, because we are growing small trees, each step can be done relatively quickly (compared to say, bagging)
- To deal with the first downside, a measure of variable importance could be used.

# Variable importance for trees

- For a single decision tree $\hat{f}^{\text{tree}}$, we define the squared importance for variable $j$ as

$$\text{Imp}_j^2(\hat{f}^{\text{tree}}) = \sum_{k=1}^{m} \hat{d}_k I(\text{split at node } k \text{ is on variable } j)$$

where $m$ is the number of internal nodes (non-leaves), and $\hat{d}_k$ is the improvement in Gini index (or RSS) from making the $k$th split.



# Variable importance for boosting/bagging

- For boosting/bagging, we define the squared importance for a variable $j$ by simply averaging the squared importance over all of the fitted trees:

$$\text{Imp}_j^2(\hat{f}^{\text{boost}}) = \frac{1}{B}\sum_{b=1}^{B}\text{Imp}_j^2(\hat{f}^{\text{tree},b})$$

- To get the importance, we just take the squared root.
- We also usually set the largest importance to 100, and scale all of the other variable importances accordingly, which are then called relative importances
- This averaging stabilizes the variable importances, which means that they tend to be much more accurate for boosting than they do for any single tree

# Bagging and boosting in R

- The package `adabag` implements both bagging and boosting (AdaBoost) for trees, via the functions `bagging` and `boosting`
- The package `gbm` implements both AdaBoost and gradient boosting
- The package `mboost` is an alternative for boosting, that is quite flexible.
- Finally, one of the hottest boosting package is `xgboost`.