05. Tree-based Methods

Table of Contents

1. Introduction to tree-based methods
2. Regression tree
   - Tree-building process
   - Recursive binary splitting algorithm
   - Pruning a tree
3. Classification tree
4. Ensemble methods
   - Bagging
   - Random forest
   - Boosting
   - Bayesian additive regression trees

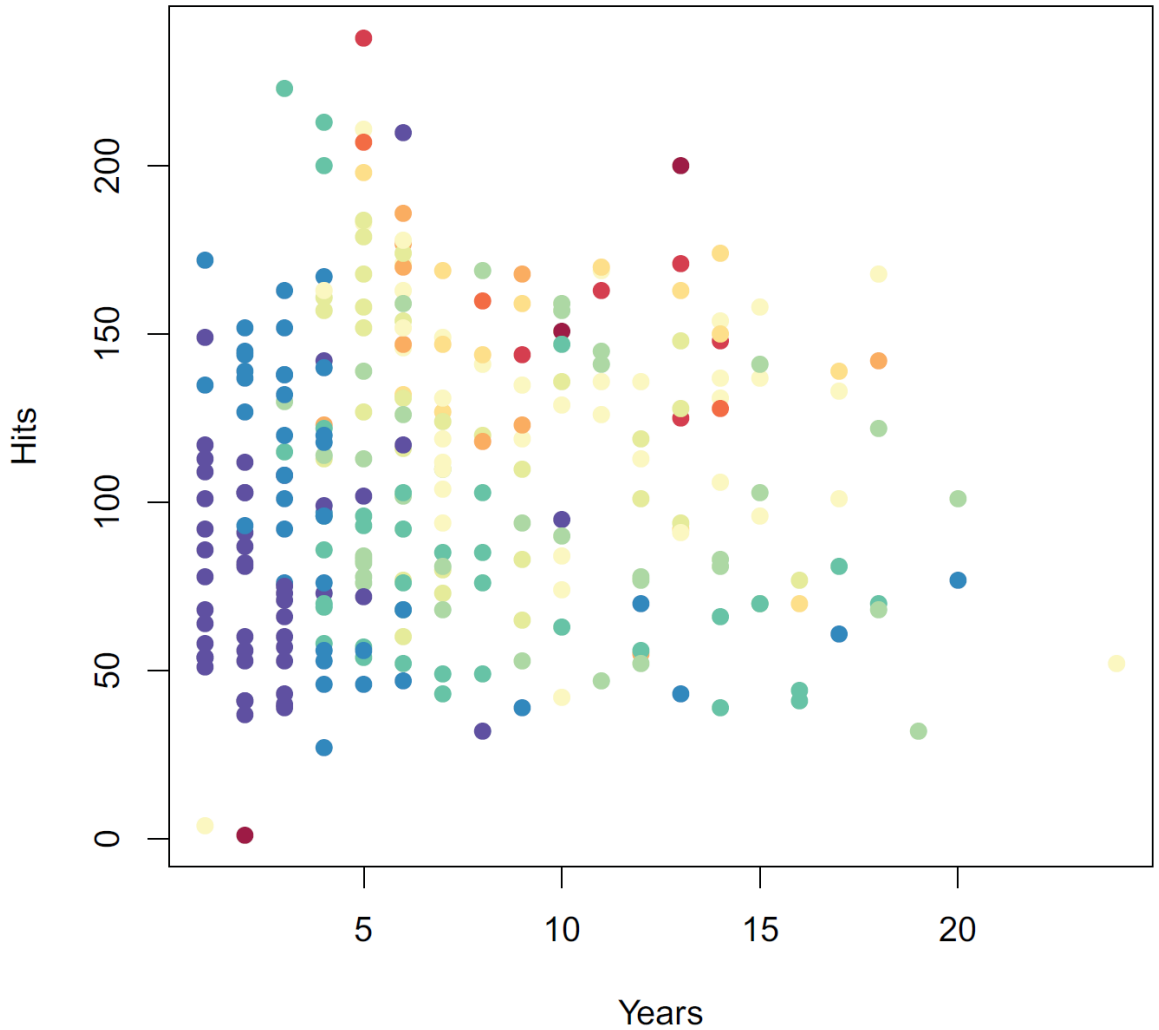
Tree-based Methods

- Tree-based methods for regression and classification involve stratifying or segmenting the predictor space into a number of simple regions.
- Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as decision-tree methods.
- Tree-based methods are simple and useful for interpretation.
- However, they typically are not competitive with the best supervised learning approaches in terms of prediction accuracy.

Tree-based Methods

- We also discuss bagging and random forests. These methods grow multiple trees which are then combined to yield a single consensus prediction.
- Combining a large number of trees can often result in dramatic improvements in prediction accuracy, at the expense of some loss interpretation.
- Decision trees can be applied to both regression and classification problems.
- We first consider regression problems, and then move on to classification.

Baseball Salary Data



- Salary is color-coded from low (blue, green) to high (yellow, red)

Decision Tree

- A regression tree for predicting the log salary of a baseball player, based on the number of years that he has played in the major leagues and the number of hits that he made in the previous year.

Interpretation of Decision Tree

- At a given internal node, the label (of the form $X_j < t_k$) indicates the left-hand branch emanating from that split, and the right-hand branch corresponds to $X_j \geq t_k$.
- For instance, the split at the top of the tree results in two large branches. The left-hand branch corresponds to `Years` $<$ `4.5`, and the right-hand branch corresponds to `Years` $\geq$ `4.5`.
- The tree has two internal nodes and three terminal nodes, or leaves.
- The number in each leaf is the mean of the response for the observations that fall there.

```
library(ISLR)
library(tree)
```

```
data(Hitters)
str(Hitters)
```

```
## Missing data
summary(Hitters$Salary)
miss <- is.na(Hitters$Salary)
```

```
## Fit a regression tree
g <- tree(log(Salary) ~ Years + Hits, subset=!miss, Hitters)
g
summary(g)
```

```
## Draw a tree
plot(g)
text(g, pretty=0)
```

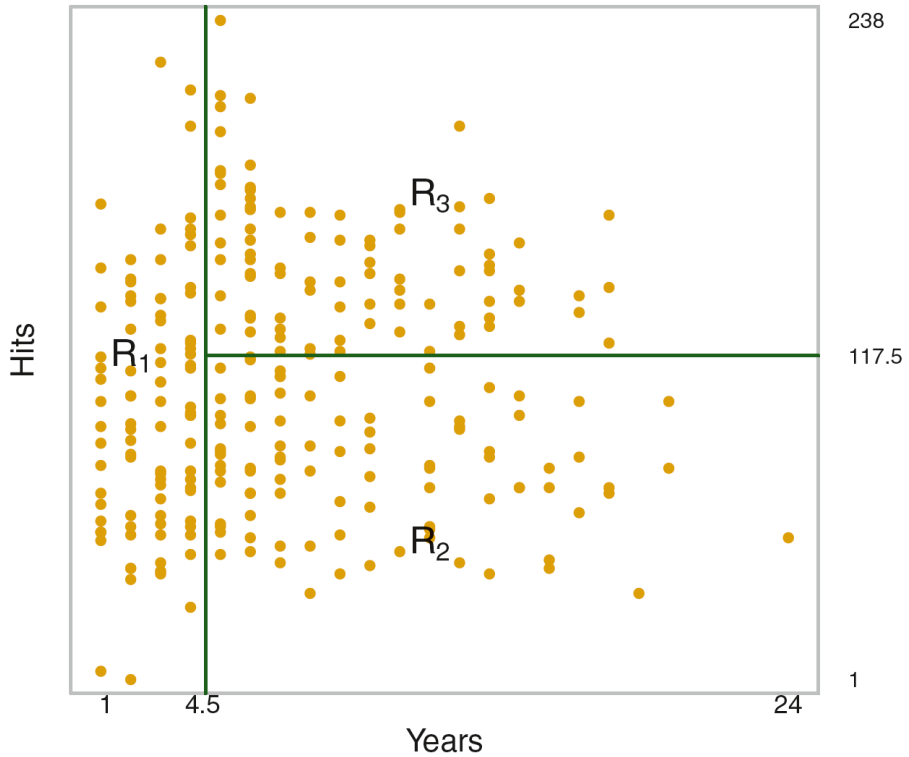
```
## Prune a tree
g2 <- prune.tree(g, best=3)
plot(g2)
text(g2, pretty=0)
```

```
## Another R package for tree
library(rpart)
library(rpart.plot)
```

```
## Fit a regression tree
g3 <- rpart(log(Salary) ~ Years + Hits, subset=!miss, Hitters)
g3
summary(g3)
```

```
## Draw a fancy tree
prp(g3, branch.lwd=4, branch.col="darkgreen", extra=101)
```

Results



- Overall, the tree stratifies or segments the players into three regions of predictor space:
  $R_1 = \{X | \texttt{Years} < 4.5\}$,
  $R_2 = \{X | \texttt{Years} \geq 4.5, \texttt{Hits} < 117.5\}$, and
  $R_3 = \{X | \texttt{Years} \geq 4.5, \texttt{Hits} \geq 117.5\}$.

```
par(mar=c(4,4,3,4))
plot(Hitters$Years, Hitters$Hits, pch=16, col="orange",
     xlab="Years", ylab="Hits", xaxt="n", yaxt="n")
axis(1, at=c(1,4.5,24), labels=c("1","4.5","24"),
     tick=FALSE, pos=1)
axis(4, at=c(1,117.5,238), labels=c("1","117.5","238"),
     tick=FALSE, las=2)
abline(v=4.5, lwd=2, col="darkgreen")
segments(4.5,117.5,25,117.5, lwd=2, lty=1, col="darkgreen")
text(2, 117.5, font=2, cex=1.4, expression(R[1]))
text(13, 50, font=2, cex=1.4, expression(R[2]))
text(13, 180.5, font=2, cex=1.4, expression(R[3]))
```

```
## Compute the mean of log Salary for each region
attach(Hitters)
mean(log(Salary[Years < 4.5]), na.rm=TRUE)
mean(log(Salary[Years >= 4.5 & Hits < 117.5]), na.rm=TRUE)
mean(log(Salary[Years >= 4.5 & Hits >= 117.5]), na.rm=TRUE)
```

Terminology for Trees

- In keeping with the tree analogy, the regions R_1, R_2, and R_3 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.
- In the hitters tree, the two internal nodes are indicated by the text `Years`<4.5 and `Hits`<117.5.

Interpretation of the Results

- `Years` is the most important factor in determining `Salary`, and players with less experience earn lower salaries than more experienced players.
- Given that a player is less experienced, the number of `Hits` that he made in the previous year seems to play little role in his `Salary`.
- But, among players who have been in the major leagues for five or more years, the number of `Hits` made in the previous year does affect `Salary`, and players who made more `Hits` last year tend to have higher salaries.
- Surely an over-simplification, but compared to a regression model, it is easy to display, interpret and explain.

Details of the Tree-building Process

- We divide the predictor space – that is, the set of possible values for $X_1, X_2, \ldots, X_p$ – into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$.
- For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.
- In theory, the regions could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles, or boxes, for simplicity and for ease of interpretation of the resulting predictive model.

Details of the Tree-building Process

- The goal is to find boxes $R_1, \ldots, R_J$ that minimize the RSS, given by

$$\text{RSS} = \sum_{j=1}^{J} \sum_{i \in R_j} \left(y_i - \hat{y}_{R_j} \right)^2,$$

  where $\hat{y}_{R_j}$ is the mean response for the training observations within the jth box.
- Unfortunately, it is computationally infeasible to consider every possible partition of the feature space into J boxes.

More Details of the Tree-building Process

- We take a top-down, greedy approach that is known as recursive binary splitting.
- The approach is top-down because it begins at the top of the tree and then successively splits the predictor space; each split is indicated via two new branches further down on the tree.
- It is greedy because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

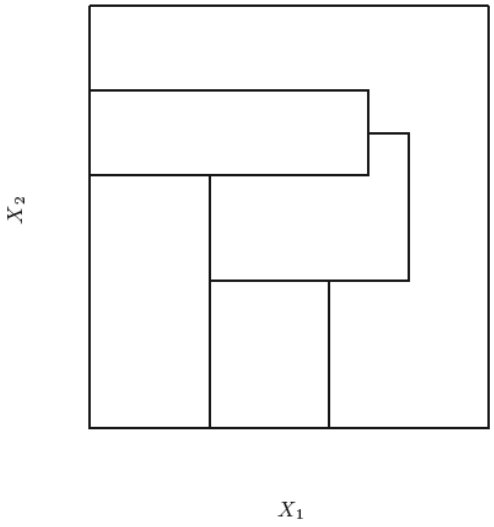
Recursive Binary Splitting

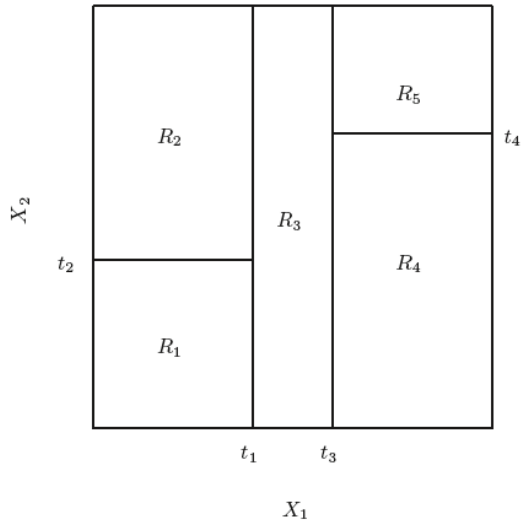
- We first select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the greatest possible reduction in RSS.
- Next, we repeat the process, looking for the best predictor and best cutpoint in order to split the data further so as to minimize the RSS within each of the resulting regions.
- However, this time, instead of splitting the entire predictor space, we split one of the two previously identified regions. We now have three regions.
- Again, we look to split one of these three regions further, so as to minimize the RSS. The process continues until a stopping criterion is reached; for instance, we may continue until no region contains more than five observations.

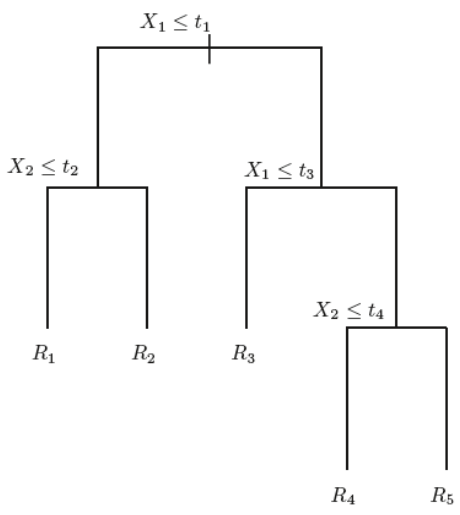
Predictions

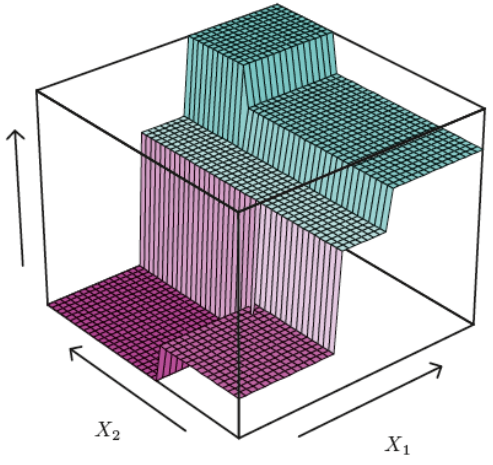
- We predict the response for a given test observation using the mean of the training observations in the region to which that test observation belongs.
- A five-region example of this approach is shown in the next slide.
  - Top left: A partition of two-dimensional feature space that could not result from recursive binary splitting.
  - Top right: The output of recursive binary splitting on a two-dimensional example.
  - Bottom Left: A tree corresponding to the partition in the top right panel.
  - Bottom Right: A perspective plot of the prediction surface corresponding to that tree.

Tree Regions









Pruning a Tree

- The process described above may produce good predictions on the training set, but is likely to overfit the data, leading to poor test set performance.
- A smaller tree with fewer splits (that is, fewer regions $R_1, \ldots, R_J$) might lead to lower variance and better interpretation at the cost of a little bias.
- One possible alternative to the process described above is to grow the tree only so long as the decrease in the RSS due to each split exceeds some (high) threshold.
- This strategy will result in smaller trees, but is too short-sighted: a seemingly worthless split early on in the tree might be followed by a very good split – that is, a split that leads to a large reduction in RSS later on.
- A better strategy is to grow a very large tree T_0, and then prune it back in order to obtain a subtree.

Pruning a Tree

- Cost complexity pruning — also known as weakest link pruning — is used to do this.
- We consider a sequence of trees indexed by a nonnegative tuning parameter α. For each value of α there corresponds a subtree $T \subset T_0$ such that

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

is as small as possible. Here $|T|$ indicates the number of terminal nodes of the tree T, R_m is the rectangle (i.e. the subset of predictor space) corresponding to the mth terminal node, and $\hat{y}_{R_m}$ is the mean of the training observations in R_m.

Choosing the Best Subtree

- The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data.
  - When $\alpha = 0$, the subtree is simply equal to T_0.
  - As α increases, the quantity above will tend to be minimized for a smaller subtree.
- As we increase α from zero, branches get pruned from the tree in a nested and predictable fashion, so obtaining the whole sequence of subtrees as a function of α is easy.
- We select an optimal value $\hat{\alpha}$ using cross-validation.
- We then return to the full data set and obtain the subtree corresponding to $\hat{\alpha}$.

Summary: Tree Algorithm

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.
3. Use K-fold cross-validation to choose α. For each $k = 1, \ldots, K$:
   - (3.1) Repeat Steps 1 and 2 on the $\frac{K-1}{K}$th fraction of the training data, excluding the kth fold.
   - (3.2) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

   Average the results, and pick α to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

Hitters Data Continued

- First, we randomly divided the data set in half, yielding 132 observations in the training set and 131 observations in the test set.
- We then built a large regression tree on the training data and varied α in in order to create subtrees with different numbers of terminal nodes.
- Finally, we performed six-fold cross-validation in order to estimate the cross-validated MSE of the trees as a function of α.

```
library(tree)
data(Hitters)
```

```
miss <- is.na(Hitters$Salary)
g <- tree(log(Salary) ~ Years + Hits + RBI + PutOuts + Walks +
          Runs + Assists + HmRun + Errors + AtBat,
          subset=!miss, Hitters)
plot(g)
text(g, pretty=0)
summary(g)
```

```
## Perform 6 fold CV
set.seed(1234)
cv.g <- cv.tree(g, K=6)
plot(cv.g$size, cv.g$dev, type="b")
```

```
## Find the optimal tree size that minimizes MSE
w <- which.min(cv.g$dev)
g2 <- prune.tree(g, best=cv.g$size[w])
plot(g2)
text(g2, pretty=0)
```

```
attach(Hitters)
newdata <- data.frame(Salary, Years, Hits, RBI, PutOuts, Walks,
                      Runs, Assists, HmRun, Errors, AtBat)
newdata <- newdata[!miss,]
```

```
## Separate samples into 132 training sets and 131 test sets
set.seed(1111)
train <- sample(1:nrow(newdata), ceiling(nrow(newdata)/2))
```

```
## Fit a tree with training set and compute test MSE
tree.train <- tree(log(Salary) ~ ., subset=train, newdata)
yhat1 <- exp(predict(tree.train, newdata[-train, ]))
tree.test <- newdata[-train, "Salary"]
plot(yhat1, tree.test)
abline(0,1)
sqrt(mean((yhat1-tree.test)^2))
```

```
## Perform 6 fold CV for training sets
set.seed(1234)
cv.g <- cv.tree(tree.train, K=6)
plot(cv.g$size, cv.g$dev, type="b")
```

```
## Prune a tree with training set and compute test MSE
## in the original sclae
w <- which.min(cv.g$dev)
prune.tree <- prune.tree(tree.train, best=cv.g$size[w])
yhat2 <- exp(predict(prune.tree, newdata[-train, ]))
plot(yhat2, tree.test)
abline(0,1)
sqrt(mean((yhat2-tree.test)^2))
```

```
## Compute test MSE of least square estimates
g0 <- lm(log(Salary)~., newdata, subset=train)
yhat3 <- exp(predict(g0, newdata[-train,]))
sqrt(mean((yhat3-newdata$Salary[-train])^2))
```

```
library(MASS)
data(Boston)
?Boston
str(Boston)
```

```
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston)/2)
tree.boston <- tree(medv ~ ., Boston, subset=train)
summary(tree.boston)
plot(tree.boston)
text(tree.boston, pretty=0)
```

```
cv.boston <- cv.tree(tree.boston, K=5)
plot(cv.boston$size, cv.boston$dev, type="b")
which.min(cv.boston$dev)
```

```
yhat <- predict(tree.boston, newdata=Boston[-train, ])
boston.test <- Boston[-train, "medv"]
plot(yhat, boston.test)
abline(0, 1)
mean((yhat - boston.test)^2)
```

```
g <- lm(medv ~ ., Boston, subset=train)
pred <- predict(g, Boston[-train,])
mean((pred - boston.test)^2)
```

```
library(leaps)
g1 <- regsubsets(medv ~ ., data=Boston, nvmax=13, subset=train)
ss <- summary(g1)
cr <- cbind(ss$adjr2, ss$cp, ss$bic)
x.test <- as.matrix(Boston[-train, -14])
MSE <- NULL
for (i in 1:3) {
    beta <- rep(0, ncol(Boston))
    if (i > 1) ww <- which.min(cr[,i])
    else ww <- which.max(cr[,i])
    beta[ss$which[ww,]] <- coef(g1, ww)
    preds <- cbind(1, x.test) %*% beta
    MSE[i] <- mean((preds - boston.test)^2)
}
MSE
```

Classification Tree

- Very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one.
- For a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs.
- Just as in the regression setting, we use recursive binary splitting to grow a classification tree.
- In the classification setting, RSS cannot be used as a criterion for making the binary splits.

Classification Tree

- A natural alternative to RSS is the classification error rate, which is also called misclassification rate.
- The classification error rate is simply the fraction of the training observations in that region that do not belong to the most common class:

$$\text{Error} = 1 - \max_k(\hat{p}_{mk}),$$

where $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class.

- However, classification error is not sufficiently sensitive for tree-growing, and in practice two other measures are preferable.

Gini index and Cross-entropy

- The Gini index is defined by

$$G_m = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

  a measure of total variance across the K classes. The Gini index is small if all of the $\hat{p}_{mk}$'s are close to either zero or one.
- For this reason the Gini index is referred to as a measure of node purity - a small value indicates that a node contains predominantly observations from a single class.
- An alternative to the Gini index is cross-entropy, given by

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

- Gini index and cross-entropy are very similar numerically.

Deviance of Classification Trees

- The deviance of the mth node is defined as

$$D_m = -\sum_{k=1}^{K} n_{mk} \log(\hat{p}_{mk}),$$

  where n_{mk} is the number of observations in the mth node that belong to the kth class.

- A small deviance indicates a tree that provides a good fit to the training data.

- The residual mean deviance is simply

$$\frac{2}{n - |T_0|} \sum_m D_m,$$

  where $|T_0|$ is the size of the tree.

Example: Heart Data

- These data contain a binary outcome `AHD` for 303 patients who presented with chest pain.
- An outcome value of `Yes` indicates the presence of heart disease based on an angiographic test, while `No` means no heart disease.
- There are 13 predictors including `Age`, `Sex`, `Chol` (a cholesterol measurement), and other heart and lung function measurements.
- We can find some missing values from data, so we need to remove the samples with missing values. Then, the sample size is reduced down to 297.

```
url.ht <- "https://www.statlearning.com/s/Heart.csv"
Heart <- read.csv(url.ht, h=T)
summary(Heart)
```

```
Heart <- Heart[, colnames(Heart)!="X"]
Heart[,"Sex"] <- factor(Heart[,"Sex"], 0:1, c("female", "male"))
Heart[,"Fbs"] <- factor(Heart[,"Fbs"], 0:1, c("false", "true"))
Heart[,"ExAng"] <- factor(Heart[,"ExAng"], 0:1, c("no", "yes"))
Heart[,"ChestPain"] <- as.factor(Heart[,"ChestPain"])
Heart[,"Thal"] <- as.factor(Heart[,"Thal"])
Heart[,"AHD"] <- as.factor(Heart[,"AHD"])
```

```
summary(Heart)
dim(Heart)
sum(is.na(Heart))
```

```
Heart <- na.omit(Heart)
dim(Heart)
summary(Heart)
```

```
c1 <- c("Age", "RestBP", "Chol", "RestECG", "MaxHR", "Oldpeak",
        "Slope", "Ca")
```

```
par(mfrow=c(2,4))
for (i in 1:8) {
    plot(Heart[,c1[i]], pch=20, main=c1[i],
         col=as.numeric(Heart$AHD) + 1)
}
```

```
library(ggplot2)
library(gridExtra)
g1 <-ggplot(Heart, aes(x=Sex, fill=AHD)) +
            geom_bar(position="stack")
g2 <-ggplot(Heart, aes(x=ChestPain,fill=AHD)) +
            geom_bar(position="stack")
g3 <-ggplot(Heart, aes(x=Fbs, fill=AHD)) +
            geom_bar(position="stack")
g4 <-ggplot(Heart, aes(x=ExAng, fill=AHD)) +
            geom_bar(position="stack")
g5 <-ggplot(Heart,aes(x=Thal, fill=AHD)) +
            geom_bar(position="stack")
grid.arrange(g1, g2, g3, g4, g5, nrow=2)
```

```
g <- glm(AHD ~., family="binomial", Heart)
summary(g)
```

```
library(tree)
tree.heart <- tree(AHD ~., Heart)
summary(tree.heart)
tree.heart
```

```
plot(tree.heart)
text(tree.heart)
plot(tree.heart)
text(tree.heart, pretty=0)
```

```
## predict the probability of each class or class type
predict(tree.heart, Heart)
predict(tree.heart, Heart, type="class")
```

```
## Compute classification error rate of training observations
pred <- predict(tree.heart, Heart, type="class")
table(pred, Heart$AHD)
mean(pred!=Heart$AHD)
```

```
## Separate training and test sets
set.seed(123)
train <- sample(1:nrow(Heart), nrow(Heart)/2)
test <- setdiff(1:nrow(Heart), train)
heart.test <- Heart[test, ]
heart.tran <- tree(AHD ~., Heart, subset=train)
heart.pred <- predict(heart.tran, heart.test, type="class")
```

```
## Compute classification error rate
table(heart.pred, Heart$AHD[test])
mean(heart.pred!=Heart$AHD[test])
```

```
## Run 5-fold cross validataion
set.seed(1234)
cv.heart <- cv.tree(heart.tran, FUN=prune.misclass, K=5)
cv.heart
```

```
par(mfrow=c(1, 2))
plot(cv.heart$size, cv.heart$dev, type="b")
plot(cv.heart$k, cv.heart$dev, type="b")
```

```
## Find the optimal tree size
w <- cv.heart$size[which.min(cv.heart$dev)]
```

```
## Prune the tree with the optimal size
prune.heart <- prune.misclass(heart.tran, best=w)
```

```
par(mfrow=c(1, 2))
plot(heart.tran)
text(heart.tran)
plot(prune.heart)
text(prune.heart, pretty=0)
```

```
## Compute classification error of the subtree
heart.pred <- predict(prune.heart, heart.test, type="class")
table(heart.pred, Heart$AHD[test])
mean(heart.pred!=Heart$AHD[test])
```

```
set.seed(111)
K <- 100
RES1 <- matrix(0, K, 2)
```

```
for (i in 1:K) {
    train <- sample(1:nrow(Heart), floor(nrow(Heart)*2/3))
    test <- setdiff(1:nrow(Heart), train)
    heart.test <- Heart[test, ]
    heart.tran <- tree(AHD ~., Heart, subset=train)
    heart.pred <- predict(heart.tran, heart.test, type="class")
    RES1[i,1] <- mean(heart.pred!=Heart$AHD[test])
    cv.heart <- cv.tree(heart.tran, FUN=prune.misclass, K=5)
    w <- cv.heart$size[which.min(cv.heart$dev)]
    prune.heart <- prune.misclass(heart.tran, best=w)
    heart.pred.cv <- predict(prune.heart, heart.test,
                             type="class")
    RES1[i,2] <- mean(heart.pred.cv!=Heart$AHD[test])
}
```

```
apply(RES1, 2, mean)
boxplot(RES1, col=c("orange", "lightblue"), boxwex=0.6,
        names=c("unpruned tree", "pruned tree"),
        ylab="Classification Error Rate")
```

```
library(MASS)
library(e1071)
```

```
set.seed(111)
K <- 100
RES2 <- matrix(0, K, 4)
```

```
for (i in 1:K) {
    train <- sample(1:nrow(Heart), floor(nrow(Heart)*2/3))
    test <- setdiff(1:nrow(Heart), train)
    y.test <- Heart$AHD[test]

    ## Logistic regression
    g1 <- glm(AHD~., data=Heart, family="binomial",
              subset=train)
    p1 <- predict(g1, Heart[test,], type="response")
    pred1 <- rep("No", length(y.test))
    pred1[p1 > 0.5] <- "Yes"
    RES2[i, 1] <- mean(pred1!=y.test)
```

```
    ## LDA/QDA
    g2 <- lda(AHD~., data=Heart, subset=train)
    g3 <- qda(AHD~., data=Heart, subset=train)
    pred2 <- predict(g2, Heart[test,])$class
    pred3 <- predict(g3, Heart[test,])$class
    RES2[i, 2] <- mean(pred2!=y.test)
    RES2[i, 3] <- mean(pred3!=y.test)

    ## Bayes Naive
    g4 <- naiveBayes(AHD~., data=Heart, subset=train)
    pred4 <- predict(g4, Heart[test,])
    RES2[i, 4] <- mean(pred4!=y.test)
}
```

```
apply(RES2, 2, mean)
boxplot(cbind(RES1, RES2), col=2:8, boxwex=0.6,
        names=c("unpruned tree", "pruned tree", "Logistic",
        "LDA", "QDA", "BayesNaive"),
        ylab="Classification Error Rate")
```

Advantages and Disadvantages of Trees

- Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
- Some people believe that decision trees more closely mirror human decision-making than do the regression and other classification approaches.
- Trees can be displayed graphically, and are easily interpreted even by a non-expert.
- Trees can easily handle qualitative predictors without the need to create dummy variables.
- Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches.
  - However, by aggregating many decision trees, the predictive performance of trees can be substantially improved.

Ensemble Method

- An ensemble method is an approach that combines many simple "building ensemble block" models in order to obtain a single and potentially very powerful model.
- These simple building block models are sometimes known as weak learners.
- Four ensemble methods will be discussed,
  - Bagging
  - Random forest
  - Boosting
  - Bayesian additive regression trees
- The simple building block of these ensemble method is a regression or a classification tree.

Bagging

- Bootstrap aggregation, or bagging, is a general-purpose procedure for reducing the variance of a statistical learning method; we introduce it here because it is particularly useful and frequently used in the context of decision trees.
- Recall that given a set of n independent observations

$$Z_1, Z_2, \ldots, Z_n,$$

  each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by σ^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.

Bagging

- Instead, we can bootstrap, by taking repeated samples from the (single) training data set.
- In this approach we generate B different bootstrapped training data sets. We then train our method on the bth bootstrapped training set in order to get $\hat{f}^{*b}(x)$,

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x).$$

- This is called bagging.
- For classification trees: for each test observation, we record the class predicted by each of the B trees, and take a majority vote: the overall prediction is the most commonly occurring class among the B predictions.

```
## Separate training and test sets
set.seed(123)
train <- sample(1:nrow(Heart), nrow(Heart)/2)
test <- setdiff(1:nrow(Heart), train)
```

```
## Classification error rate for single tree
heart.tran <- tree(AHD ~., subset=train, Heart)
heart.pred <- predict(heart.tran, Heart[test, ], type="class")
tree.err <- mean(Heart$AHD[test]!=heart.pred)
tree.err
```

```
## Bagging
set.seed(12345)
B <- 500
n <- nrow(Heart)
```

```
Vote <- rep(0, length(test))
bag.err <- NULL
```

```
for (i in 1:B) {
    index <- sample(train, replace=TRUE)
    heart.tran <- tree(AHD ~., Heart[index,])
    heart.pred <- predict(heart.tran, Heart[test, ], type="class")
    Vote[heart.pred=="Yes"] <- Vote[heart.pred=="Yes"] + 1
    preds <- rep("Yes", length(test))
    preds[Vote < i/2] <- "No"
    bag.err[i] <- mean(Heart$AHD[test]!=preds)
}
```

```
plot(bag.err, type="l", xlab="Number of Trees", col=1,
     ylab="Classification Error Rate")
abline(h=tree.err, lty=2, col=2)
legend("topright", c("Single tree", "Bagging"), col=c(2,1),
       lty=c(2,1))
```

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 ith observation using each of the trees in which that observation was OOB. This will yield around $B/3$ predictions for the ith observation, which we average.
- This estimate is essentially the LOO cross-validation error for bagging, if B is large.

```
## n: sample size, B: the number of bootstrap
## K: the number of simulation
n <- 100
B <- 500
K <- 10
```

```
m <- array(seq(n), c(n, B, K))
out <- matrix(0, n, K)
```

```
for (i in 1:K) {
    nm <- apply(m[,,i], 2, function(x) sample(x, replace=TRUE))
    x <- matrix(0, n, B)
    for (j in 1:B) x[nm[,j], j] <- 1
    out[,i] <- apply(x, 1, sum)/B
}
```

```
boxplot(out, boxwex=0.5, col="orange", ylim=c(0.5,0.7),
        xlab="Simulation replication",
        ylab="Proportion of bootstrapped observations")
```

```
## Average of misclassification rate for single tree
## over 50 replications
set.seed(12345)
K <- 50
Err <- NULL
```

```
for (i in 1:K) {
    train <- sample(1:nrow(Heart), nrow(Heart)*2/3)
    test <- setdiff(1:nrow(Heart), train)
    heart.tran <- tree(AHD ~., subset=train, Heart)
    heart.pred <- predict(heart.tran, Heart[test, ], type="class")
    Err[i] <- mean(Heart$AHD[test]!=heart.pred)
}
```

```
summary(Err)
Tree.Err <- mean(Err)
```

```
set.seed(1234)
Valid <- Vote <- Mis <- rep(0, nrow(Heart))
OOB.err <- NULL
```

```
for (i in 1:B) {
    index <- sample(1:nrow(Heart), replace=TRUE)
    test <- setdiff(1:nrow(Heart), unique(index))
    Valid[test] <- Valid[test] + 1
    heart.tran <- tree(AHD ~., Heart[index,])
    heart.pred <- predict(heart.tran, Heart[test,], type="class")
    Vote[test] <- Vote[test] + (heart.pred=="Yes")
    preds <- rep("Yes", length(test))
    preds[Vote[test]/Valid[test] < 0.5] <- "No"
    wh <- which(Heart$AHD[test]!=preds)
    Mis[test[wh]] <- -1
    Mis[test[-wh]] <- 1
    OOB.err[i] <- sum(Mis==-1)/sum(Mis!=0)
}
```

```
summary(OOB.err)
summary(OOB.err[-c(1:100)])
```

```
plot(OOB.err, type="l", xlab="Number of Trees", col=1,
     ylab="Classification Error Rate", ylim=c(0.1,0.4))
abline(h=Tree.Err, lty=2, col=2)
legend("topright", c("Single tree", "OOB"), col=c(2,1),
       lty=c(2,1))
```

```
## Revisit Boston data set with a quantitative response
library(MASS)
data(Boston)
summary(Boston)
dim(Boston)
```

```
## R pakcge "randomForest" fits both bagging and randomForest
library(randomForest)
bag.boston <- randomForest(medv~., data=Boston, mtry=13)
bag.boston
```

```
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston)/2)
boston.test <- Boston[-train, "medv"]
```

```
## Single Tree
stree <- tree(medv~., data=Boston, subset=train)
yhat.st <- predict(stree, newdata=Boston[-train, ])
mean((yhat.st - boston.test)^2)
```

```
## Bagging
bag <- randomForest(medv~., data=Boston, mtry=13, subset=train)
yhat.bag <- predict(bag, newdata=Boston[-train, ])
mean((yhat.bag - boston.test)^2)
```

```
bag0 <- randomForest(medv~., data=Boston, mtry=13, subset=train,
                     ntree=50)
yhat.bag0 <- predict(bag0, newdata=Boston[-train, ])
mean((yhat.bag0 - boston.test)^2)
plot(bag)
```

Random Forest

- 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 predictors.
- A fresh selection of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$. That is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors. (4 out of the 13 for the Heart data)

```
set.seed(123)
train <- sample(1:nrow(Heart), nrow(Heart)/2)
test <- setdiff(1:nrow(Heart), train)
```

```
## Bagging: m=13, Random forest: m=1, 4, 6
B <- 500
n <- nrow(Heart)
m <- c(1, 4, 6, 13)
Err <- matrix(0, B, length(m))
Vote <- matrix(0, length(test), length(m))
```

```
for (i in 1:B) {
    index <- sample(train, replace=TRUE)
    for (k in 1:length(m)) {
        s <- c(sort(sample(1:13, m[k])), 14)
        tr <- tree(AHD ~., data=Heart[index, s])
        pr <- predict(tr, Heart[test, ], type="class")
        Vote[pr=="Yes", k] <- Vote[pr=="Yes", k] + 1
```

```
        PR <- rep("Yes", length(test))
        PR[Vote[,k] < i/2] <- "No"
        Err[i, k] <- mean(Heart$AHD[test]!=PR)
    }
}
```

```
labels <- c("m = 1", "m = 4", "m = 6", "m = 13")
matplot(Err, type="l", xlab="Number of Trees", lty=1,
        col=c(1,2:4), ylab="Classification Error Rate")
legend("topright", legend=labels, col=c(1,2:4), lty=1)
```

```
colnames(Err) <- labels
apply(Err, 2, summary)
```

```
library(randomForest)
## Bagging (m=13)
bag.heart <- randomForest(x=Heart[train, -14], y=Heart[train,14],
              xtest=Heart[test,-14], ytest=Heart[test,14],
              mtry=13, importance=TRUE)
bag.heart
bag.conf <- bag.heart$test$confusion[1:2,1:2]
1- sum(diag(bag.conf))/sum(bag.conf)
```

```
## Random forest with m=1
rf1 <- randomForest(x=Heart[train,-14], y=Heart[train,14],
                    xtest=Heart[test,-14], ytest=Heart[test,14],
                    mtry=1, importance=TRUE)
rf1.conf <- rf1$test$confusion[1:2, 1:2]
1- sum(diag(rf1.conf))/sum(rf1.conf)
```

```
## Random forest with m=4
rf2 <- randomForest(x=Heart[train,-14], y=Heart[train,14],
                    xtest=Heart[test,-14], ytest=Heart[test,14],
                    mtry=4, importance=TRUE)
rf2.conf <- rf2$test$confusion[1:2,1:2]
1- sum(diag(rf2.conf))/sum(rf2.conf)
```

```
## Random forest with m=6
rf3 <- randomForest(x=Heart[train,-14], y=Heart[train,14],
                    xtest=Heart[test,-14], ytest=Heart[test,14],
                    mtry=6, importance=TRUE)
rf3.conf <- rf3$test$confusion[1:2,1:2]
1- sum(diag(rf3.conf))/sum(rf3.conf)
```

```
set.seed(1111)
N <- 50
CER <- matrix(0, N, 13)
```

```
for (i in 1:N) {
    train <- sample(1:nrow(Heart), floor(nrow(Heart)*2/3))
    test <- setdiff(1:nrow(Heart), train)
    for (k in 1:13) {
        rf <- randomForest(x=Heart[train, -14],
                  y=Heart[train, 14], xtest=Heart[test, -14],
                  ytest=Heart[test, 14], mtry=k)
        rfc <- rf$test$confusion[1:2,1:2]
        CER[i, k] <- 1-sum(diag(rfc))/sum(rfc)
    }
}
```

```
apply(CER, 2, mean)
boxplot(CER, boxwex=0.5, main="Random Forest with m = 1 to 13",
        ylab="Classification Error Rates", col="orange",
        ylim=c(0, 0.4))
```

```
varImpPlot(rf1)
varImpPlot(rf2)
varImpPlot(rf3)
```

```
(imp1 <- importance(rf1))
(imp2 <- importance(rf2))
(imp3 <- importance(rf3))
```

```
par(mfrow=c(1,3))
barplot(sort(imp1[,3]), main="RF (m=1)", horiz=TRUE, col=2)
barplot(sort(imp2[,3]), main="RF (m=4)", horiz=TRUE, col=2)
barplot(sort(imp3[,3]), main="RF (m=6)", horiz=TRUE, col=2)
```

```
barplot(sort(imp1[,4]), main="RF (m=1)", horiz=TRUE, col=2)
barplot(sort(imp2[,4]), main="RF (m=4)", horiz=TRUE, col=2)
barplot(sort(imp3[,4]), main="RF (m=6)", horiz=TRUE, col=2)
```

Boosting

- Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification. We only discuss boosting for decision trees.
- Recall that bagging involves creating multiple copies of the original training data set using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model.
- Boosting works in a similar way, except that the trees are grown sequentially: each tree is grown using information from previously grown trees.
- Note that in bagging each tree is built on a bootstrap data set, independent of the other trees.

Boosting Algorithm

Algorithm 8.2 *Boosting for Regression Trees*

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \ldots, B$, repeat:
   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).
   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{8.10}$$

   (c) Update the residuals,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$

3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8.12}$$

- Boosting for classification is similar in spirit to boosting for regression, but is a bit more complex.

Boosting

- Unlike fitting a single large decision tree to the data, the boosting approach instead learns slowly.
- Given the current model, we fit a decision tree to the residuals from the model. We then add this new decision tree into the fitted function in order to update the residuals.
- Each of these trees can be rather small, with just a few terminal nodes, determined by the parameter d in the algorithm.
- By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well. The shrinkage parameter λ slows the process down even further, allowing more and different shaped trees to attack the residuals.

Tuning parameters for boosting

- The number of trees B. Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all.
- The shrinkage parameter λ, a small positive number. This controls the rate at which boosting learns. Typical values are 0.01 or 0.001, and the right choice can depend on the problem. Very small λ can require using a very large value of B in order to achieve good performance.
- The number of splits d in each tree, which controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a stump, consisting of a single split and resulting in an additive model. More generally d is the interaction depth, and controls the interaction order of the boosted model, since d splits can involve at most d variables.

```
library(gbm)
```

```
set.seed(123)
train <- sample(1:nrow(Heart), nrow(Heart)/2)
test <- setdiff(1:nrow(Heart), train)
```

```
## Create (0,1) response
Heart0 <- Heart
Heart0[,"AHD"] <- as.numeric(Heart$AHD)-1
```

```
## boosting (d=1)
boost.d1 <- gbm(AHD~., data=Heart0[train, ], n.trees=1000,
                distribution="bernoulli", interaction.depth=1)
summary(boost.d1)
```

```
yhat.d1 <- predict(boost.d1, newdata=Heart0[test, ],
                   type="response", n.trees=1000)
phat.d1 <- rep(0, length(yhat.d1))
phat.d1[yhat.d1 > 0.5] <- 1
mean(phat.d1!=Heart0[test, "AHD"])
```

```
## boosting (d=2)
boost.d2 <- gbm(AHD~., data=Heart0[train, ], n.trees=1000,
                distribution="bernoulli", interaction.depth=2)
yhat.d2 <- predict(boost.d2, newdata=Heart0[test, ],
                   type="response", n.trees=1000)
phat.d2 <- rep(0, length(yhat.d2))
phat.d2[yhat.d2 > 0.5] <- 1
mean(phat.d2!=Heart0[test, "AHD"])
```

```
## boosting (d=3)
boost.d3 <- gbm(AHD~., data=Heart0[train, ], n.trees=1000,
                distribution="bernoulli", interaction.depth=3)
yhat.d3 <- predict(boost.d3, newdata=Heart0[test, ],
                   type="response", n.trees=1000)
phat.d3 <- rep(0, length(yhat.d3))
phat.d3[yhat.d3 > 0.5] <- 1
mean(phat.d3!=Heart0[test, "AHD"])
```

```
## boosting (d=4)
boost.d4 <- gbm(AHD~., data=Heart0[train, ], n.trees=1000,
                distribution="bernoulli", interaction.depth=4)
yhat.d4 <- predict(boost.d4, newdata=Heart0[test, ],
                   type="response", n.trees=1000)
phat.d4 <- rep(0, length(yhat.d4))
phat.d4[yhat.d4 > 0.5] <- 1
mean(phat.d4!=Heart0[test, "AHD"])
```

```
## Simulation: Boosting with d=1, 2, 3 and 4
## The number of trees: 1 to 3000
set.seed(1111)
Err <- matrix(0, 3000, 4)
```

```
for (k in 1:4) {
    boost <- gbm(AHD~., data=Heart0[train, ], n.trees=3000,
                 distribution="bernoulli", interaction.depth=k)
```

```
    for (i in 1:3000) {
        yhat <- predict(boost, newdata=Heart0[test, ],
                        type="response", n.trees=i)
        phat <- rep(0, length(yhat))
        phat[yhat > 0.5] <- 1
        Err[i,k] <- mean(phat!=Heart0[test, "AHD"])
    }
}
```

```
labels <- c("d = 1", "d = 2", "d = 3", "d = 4")
matplot(Err, type="l", xlab="Number of Trees", lty=2, col=1:4,
        ylab="Classification Error Rate")
legend("topright", legend=labels, col=1:4, lty=1)
```

```
colnames(Err) <- labels
apply(Err, 2, summary)
apply(Err[-c(1:100),], 2, summary)
```

Bayesian Additive Regression Trees

- Bayesian additive regression trees (BART) is another ensemble method that uses decision trees as its building blocks.
- BART method combines other ensemble methods.
  - Each tree is constructed in a random manner as in bagging and random forests.
  - Each tree tries to capture signals not yet accounted for by the current model as in boosting.
- BART method can be viewed as a Bayesian approach
  - Each time we randomly perturb a tree in order to fit the residuals, we are in fact drawing a new tree from a posterior distribution.
  - BART algorithm can be viewed as a Markov chain Monte Carlo (MCMC) algorithm.
  - Remove out prediction results at burn-in period.

Bayesian Additive Regression Trees

- $\hat{f}_k^b(x)$ represents the prediction at x for the kth tree used in the bth iteration, where $k = 1, \ldots, K$ and $b = 1, \ldots, B$.
- In subsequent iterations, BART updates each of the K trees, one at a time.
- BART randomly chooses a perturbation to the tree from the previous iteration $\hat{f}_k^{b-1}(x)$.
- There are two components of the perturbation:
  - Change the structure of the tree by adding or pruning branches.
  - Change the prediction in each terminal node of the tree.
- In BART, there are 3 tuning parameters:
  - K: the number of trees, e.g. $K = 200$,
  - B: the number of iterations, e.g. $B = 1000$,
  - L: the number of burn-in iterations, e.g. $L = 100$.

BART Algorithm

Algorithm 8.3 *Bayesian Additive Regression Trees*

1. Let $\hat{f}_1^1(x) = \hat{f}_2^1(x) = \cdots = \hat{f}_K^1(x) = \frac{1}{nK}\sum_{i=1}^n y_i$.
2. Compute $\hat{f}^1(x) = \sum_{k=1}^K \hat{f}_k^1(x) = \frac{1}{n}\sum_{i=1}^n y_i$.
3. For $b = 2, \ldots, B$:
   (a) For $k = 1, 2, \ldots, K$:
      i. For $i = 1, \ldots, n$, compute the current partial residual
      $$r_i = y_i - \sum_{k'<k} \hat{f}_{k'}^b(x_i) - \sum_{k'>k} \hat{f}_{k'}^{b-1}(x_i).$$
      ii. Fit a new tree, $\hat{f}_k^b(x)$, to r_i, by randomly perturbing the kth tree from the previous iteration, $\hat{f}_k^{b-1}(x)$. Perturbations that improve the fit are favored.
   (b) Compute $\hat{f}^b(x) = \sum_{k=1}^K \hat{f}_k^b(x)$.
4. Compute the mean after L burn-in samples,
$$\hat{f}(x) = \frac{1}{B-L}\sum_{b=L+1}^{B} \hat{f}^b(x).$$

```
library(BART)
```

```
set.seed(123)
train <- sample(1:nrow(Heart), nrow(Heart)/2)
test <- setdiff(1:nrow(Heart), train)
x <- Heart[, -14]
y <- as.numeric(Heart[, 14])-1
xtrain <- x[train, ]
ytrain <- y[train]
xtest <- x[-train, ]
ytest <- y[-train]
```

```
## logistic BART
set.seed(11)
fit1 <- lbart(xtrain, ytrain, x.test=xtest)
names(fit1)
```

```
prob1 <- rep(0, length(ytest))
prob1[fit1$prob.test.mean > 0.5] <- 1
mean(prob1!=ytest)
```

```
## Probit BART
set.seed(22)
fit2 <- pbart(xtrain, ytrain, x.test=xtest)
prob2 <- rep(0, length(ytest))
prob2[fit2$prob.test.mean > 0.5] <- 1
mean(prob2!=ytest)
```

```
cbind(fit1$prob.test.mean, fit2$prob.test.mean)
plot(fit1$prob.test.mean, fit2$prob.test.mean, col=ytest+2,
     xlab="Logistic BART", ylab="Probit BART")
abline(0, 1, lty=3, col="grey")
abline(v=0.5, lty=1, col="grey")
abline(h=0.5, lty=1, col="grey")
legend("topleft", col=c(2,3), pch=1,
       legend=c("AHD = No", "AHD = Yes"))
```

```
## Training and test errors for both logistic and probit models
## First, make a cumulative average function
cum.avg <- function(data) {
    res <- matrix(0, nrow(data), ncol(data))
    for (i in 1:ncol(data)) {
        res[,i] <- cumsum(data[,i])/seq(1, nrow(data))
    }
return(res)
}
```

```
n <- 1000
tr1 <- cum.avg(fit1$prob.train)
tr2 <- cum.avg(fit2$prob.train)
tr3 <- cum.avg(fit1$prob.test)
tr4 <- cum.avg(fit2$prob.test)
p1 <- p2 <- matrix(0, n, ncol(tr1))
p3 <- p4 <- matrix(0, n, ncol(tr3))
p1[tr1 > 0.5] <- p2[tr2 > 0.5] <- 1
p3[tr3 > 0.5] <- p4[tr4 > 0.5] <- 1
```

```
ERR <- matrix(0, n, 4)
trm <- matrix(ytrain, n, length(ytrain), byrow=T)
tem <- matrix(ytest, n, length(ytest), byrow=T)
for (i in 1:4) {
    p0 <- get(paste("p",i,sep=""))
    if (i < 3) ERR[,i] <- apply(p0!=trm, 1, mean)
    else ERR[,i] <- apply(p0!=tem, 1, mean)
}
```

```
matplot(ERR, type="l", ylim=c(0, 0.3), lty=c(2,2,1,1),
        col=c(1,2,1,2), xlab="Number of Iterations",
        ylab="Classification Error")
legend("topright", col=c(1,2,1,2), lty=c(2,2,1,1),
       legend=c("Logistic training", "Probit training",
                "Logistic test", "Probit test"))
```

```
## Revisit Boston data set with a quantitative response
library(MASS)
summary(Boston)
dim(Boston)
```

```
set.seed(111)
train <- sample(1:nrow(Boston), floor(nrow(Boston)*2/3))
boston.test <- Boston[-train, "medv"]
```

```
## Regression tree
library(tree)
tree.boston <- tree(medv ~ ., Boston, subset=train)
yhat <- predict(tree.boston, newdata=Boston[-train, ])
mean((yhat - boston.test)^2)
```

```
## LSE: least square estimates
g0 <- lm(medv ~ ., Boston, subset=train)
pred0 <- predict(g0, Boston[-train,])
mean((pred0 - boston.test)^2)
```

```
## Bagging
library(randomForest)
g1 <- randomForest(medv ~ ., data=Boston, mtry=13, subset=train)
yhat1 <- predict(g1, newdata=Boston[-train, ])
mean((yhat1 - boston.test)^2)
```

```
## Random Forest (m = 4)
g2 <- randomForest(medv ~ ., data=Boston, mtry=4, subset=train)
yhat2 <- predict(g2, newdata=Boston[-train, ])
mean((yhat2 - boston.test)^2)
```

```
## Boosting (d = 4)
library(gbm)
g3 <- gbm(medv~., data = Boston[train, ], distribution="gaussian",
          n.trees=5000, interaction.depth=4)
yhat3 <- predict(g3, newdata=Boston[-train, ], n.trees=5000)
mean((yhat3 - boston.test)^2)
```

```
library(BART)
g4 <- gbart(Boston[train, 1:13], Boston[train, "medv"],
            x.test=Boston[-train, 1:13])
yhat4 <- g4$yhat.test.mean
mean((yhat4 - boston.test)^2)
```

```
## Simulation: 4 ensemble methods
set.seed(1111)
N <- 20
```

```
ERR <- matrix(0, N, 4)
for (i in 1:N) {
    train <- sample(1:nrow(Boston), floor(nrow(Boston)*2/3))
    boston.test <- Boston[-train, "medv"]

    ## Bagging
    g1 <- randomForest(medv ~ ., data=Boston, mtry=13,
                       subset=train)
    yhat1 <- predict(g1, newdata=Boston[-train, ])
    ERR[i,1] <- mean((yhat1 - boston.test)^2)
```

```
    ## Random forest
    g2 <- randomForest(medv ~ ., data=Boston, mtry=4,
                       subset=train)
    yhat2 <- predict(g2, newdata=Boston[-train, ])
    ERR[i, 2] <- mean((yhat2 - boston.test)^2)

    ## Boosting
    g3 <- gbm(medv~., data = Boston[train, ], n.trees=5000,
              distribution="gaussian", interaction.depth=4)
    yhat3 <- predict(g3, newdata=Boston[-train, ], n.trees=5000)
    ERR[i, 3] <- mean((yhat3 - boston.test)^2)

    ## BART
    invisible(capture.output(g4 <- gbart(Boston[train, 1:13],
              Boston[train, "medv"],
              x.test=Boston[-train, 1:13])))
    yhat4 <- g4$yhat.test.mean
    ERR[i, 4] <- mean((yhat4 - boston.test)^2)
}
```

```
labels <- c("Bagging", "RF", "Boosting", "BART")
boxplot(ERR, boxwex=0.5, main="Ensemble Methods", col=2:5,
        names=labels, ylab="Mean Squared Errors", ylim=c(0,30))
```

```
colnames(ERR) <- labels
apply(ERR, 2, summary)
apply(ERR, 2, var)
```

```
RA <- t(apply(ERR, 1, rank))
RA
apply(RA, 2, table)
```