High Dimensional Data Analysis

Bagging and random forest

Gianna Monti

Ensemble methods

Ensemble methods are techniques that create multiple models and then combine them to produce improved results.

These models, when used as inputs of ensemble methods, are often called **weak learners**.

Ensemble learning is appealing because that it is able to boost weak learners which are slightly better than random guess to strong learners which can make very accurate predictions. However, ensemble methods increases **computation time** and reduces **interpretability**.

For example, classification and regression trees are simple and useful for interpretation. However they are typically not competitive with other approaches in terms of prediction accuracy.

We will see that ensemble methods such as **bagging** and **random forests** grow multiple trees which are then combined to yield a single prediction. Combining a large number of trees can often result in dramatic improvements in prediction accuracy, at the expense of some interpretation loss.

Instability of trees

- The primary disadvantage of trees is that they are rather unstable (high variance)
- In other words, a small change in the data often results in a completely different tree
- One major reason for this instability is that if a split changes, all the splits under it change as well, thereby propagating the variability
- We will learn how to control the variance, or **stabilize** the predictions made by trees
- In doing so, we can greatly improve prediction accuracy but we suffer in terms of interpretability

The boostrap

- The **bootstrap** is an useful resampling tool in statistics
- A bootstrap sample of size n from the **training data** is

$$(\tilde{x}_1, \tilde{y}_1), (\tilde{x}_2, \tilde{y}_2), \dots, (\tilde{x}_n, \tilde{y}_n)$$

where each $(\tilde{x}_i, \tilde{y}_i)$ are drawn from uniformly at random from

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, x_n)$$

with **replacement**

- Not all of the training points are represented in a bootstrap sample, and some are represented more than once. The probability for one observation not to be drawn in one draw is $1 - \frac{1}{n}$
- For large n, the probability for one observation not to be drawn in any of the n draws is

$$\lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e} \approx 0.368$$

- We can expect $\approx 1/3$ of the n original observations to be **out-of-bag** (OOB)

```
rm(list = ls())
n <- 1000
original <- 1:n
set.seed(123)
bagged <- sample(original, size = n, replace = TRUE)
# proportion of OOB observations
length(setdiff(original, bagged))/n
```

```
[1] 0.362
```

```
# frequency
table(table(bagged))
```

```

  1   2   3   4   5 
379 178  61  18   2 
```

We refit the model to each of the bootstrap data sets, and examine the behavior of the fits over the B replications. The mean prediction error is then

$$\widehat{\text{Err}}_{Boot} = \frac{1}{B}\frac{1}{n}\sum_{b=1}^{B}\sum_{i=1}^{n}(y_i - \hat{f}^b(x_i))^2$$

with $\hat{f}^b$ indicating the function assessed on sample b and $\hat{f}^b(x_i)$ indicating the prediction of observation x_i of the bth data set.

Problem and possible improvements: Due to the large overlap in test and training sets, $\widehat{\mathrm{Err}}_{Boot}$ is frequently too optimistic. The probability of an observation being included in a bootstrap data set is $1 - (1 - 1/n)^n \approx 1 - e^{-1} = 0.632$. A possible improvement would be to calculate $\widehat{\mathrm{Err}}_{Boot}$ only for those observations not included in the bootstrap data set, which is true for about 1/3 of the observations.

"Leave-one-out bootstrap" offers another potential improvement: The prediction $\hat{f}^b(x_i)$ is based only on the data sets which do not include x_i:

$$\widehat{\mathrm{Err}}_{Boot-1} = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{|C^{-i}|} \sum_{b \in C^{-i}} (y_i - \hat{f}^b(x_i))^2$$

where C^{-i} is the set of indices of the bootstrap samples b that do not contain observation i.

Bagging

Bagging trees

- Bootstrap AGGregation, or **bagging** (Breiman (1996), “Bagging Predictors”), is a general procedure for reducing the variance of a model and it is particularly useful in the case of regression or classification trees

1. Generate B different bootstrapped training sets

$$(\tilde{x}_1^b, \tilde{y}_1^b), (\tilde{x}_2^b, \tilde{y}_2^b), \ldots, (\tilde{x}_n^b, \tilde{y}_n^b), \qquad b = 1, \ldots, B$$

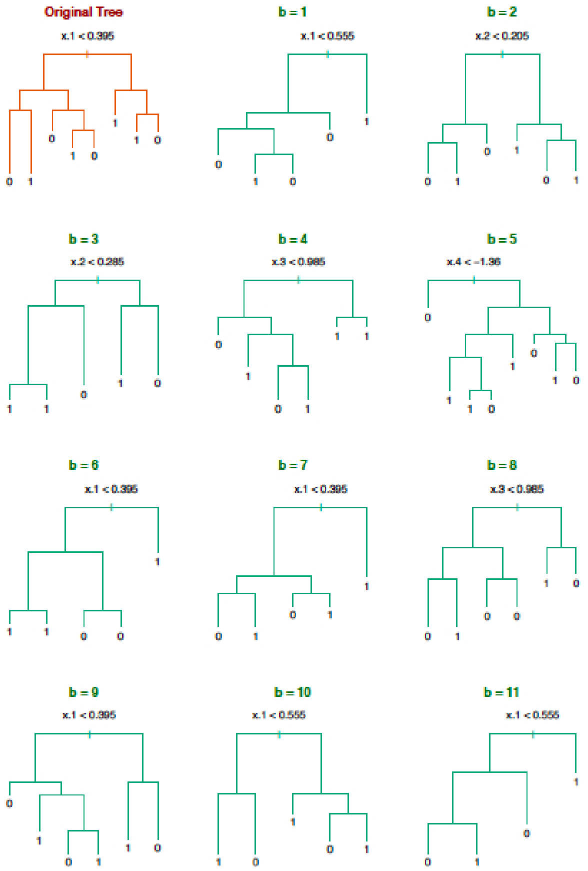
2. Fit a regression tree $\hat{f}^b$ or a classification tree $\hat{c}^b$ for each bootstrapped training set
3. Average all the predictions:

$$\bar{f}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x)$$

for regression trees and

$$\bar{c}(x) = \text{Mode}\{\hat{c}^b(x), b = 1, \ldots, B\}$$

for classification trees (**consensus**)



Source: Hastie, Tibshirani and Friedman (2009) The Elements of Statical Learning: Data Mining, Inference, and Prediction (ESL) p. 284: $n = 30$ training data points, $p = 5$ predictors, and $K = 2$ classes. No pruning used in growing trees.

Breiman (1996) Bagging predictors. *Machine Learning*

Example from the original Breiman paper on bagging: comparing the misclassification error of the CART tree (pruning performed by cross-validation) and of the bagging classifier (with $B = 50$ bootstrap samples):

Data set	CART	Bagging	Decrease
waveform	29.1	19.3	34%
heart	4.9	2.8	43%
breast cancer	5.9	3.7	37%
ionosphere	11.2	7.9	29%
diabetes	25.3	23.9	6%
glass	30.4	23.6	22%
soybean	8.6	6.8	21%

Bagging estimates of class probabilities

For a classification problem with K classes, we may want to estimate the class probabilities out of our bagging procedure.

What about using the proportion of votes that were for each class? This is generally not a good estimate!

What's nice about trees is that classification trees already gives us a set of predicted class probabilities at x: $\hat{p}_k^b(x), k = 1, \ldots, K$. These are simply the proportion of points in the appropriate region that are in each class.

This suggests an alternative method for bagging. Now given an input $x \in \mathbb{R}^p$, instead of simply taking the prediction $\hat{f}^b(x)$ from each tree, we go further and look at its predicted class probabilities $\hat{p}_k^b(x)$, $k = 1, \ldots, K$.

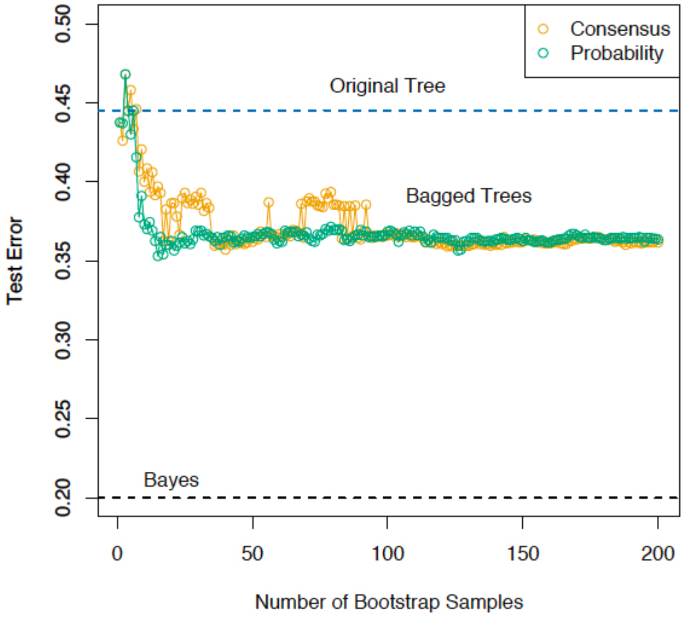
The **bagging estimates of class probabilities** are given by

$$\bar{p}_k(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{p}_k^b(x), \quad k = 1, \ldots, K$$

The final bagged classifier just chooses the **class with the highest probability**.

$$\hat{f}(x) = \arg \max_{k=1,\ldots,K} \bar{p}_k(x)$$

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.



Source: ESL p. 285: Bagging helps decrease the misclassification rate of the classifier (evaluated on a large independent test set) - orange curve. The alternative form of bagging produces misclassification errors shown in green.

Why is bagging working?

Why is bagging working? Here is a simplified setup with $K = 2$ classes "1" and "2" to help understand the basic phenomenon.

Suppose that for a given x, we have an odd number B of independent classifiers $\hat{c}^b(x), b = 1, \ldots, B$, and each classifier has a misclassification rate of 40%. Assume without loss of generality that the true class at x is "1", so

$$\Pr(\hat{c}^b(x) = 2) = 0.4$$

Let $Z = \sum_{b=1}^{B} I\{\hat{c}^b(x) = 2\}$ be the number of votes for class "2", and note that

$$Z \sim \text{Binomial}(B, 0.4)$$

Therefore the **misclassification rate** of the bagged classifier is

$$\Pr(\vec{c}^b(x) = 2) = \Pr(Z \geq (B + 1)/2)$$

which goes $\to 0$ as $B \to \infty$. In other words, the bagged classifier has perfect predictive accuracy as the number of sampled data sets $B \to \infty$.

So why did the prediction error seem to level off in our examples? Of course, the caveat here is **independence**. The classifiers that we use in practice, $\hat{c}^b$ are clearly not independent, because they are fit on very similar data sets (bootstrap samples from the same training set).

Wisdom of crowds

The wisdom of crowds is a concept popularized outside of statistics to describe the same phenomenon. It is the idea that the collection of knowledge of an **independent** group of people can exceed the knowledge of any one person individually.

Assume that we have an odd number $n > 1$ of independent individuals, and each individual has probability π of making a correct guess

The number of correct guesses is

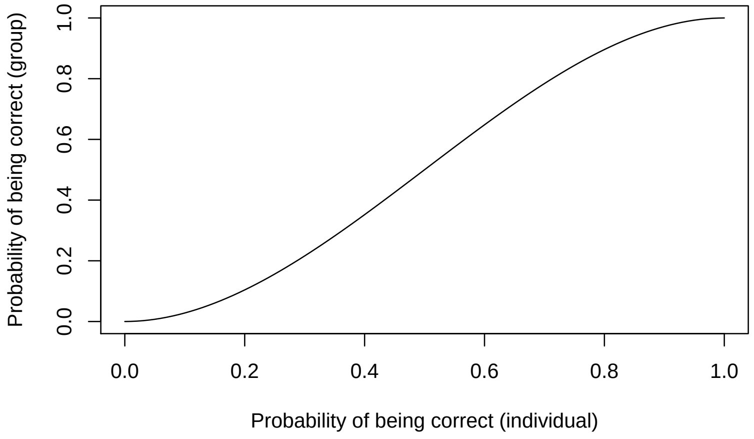
$$Z \sim \text{Binomial}(n, \pi)$$

The majority vote makes the correct guess when $Z \geq z$ with $z = (n+1)/2$, i.e.

$$\Pr(Z \geq z) = \sum_{k=z}^{n} \binom{n}{k} \pi^k (1-\pi)^{n-k}$$

e.g. if $n = 3$ and $\pi = 70\%$, then the majority vote makes the correct guess with probability 78.4%.

```
rm(list = ls())
pi <- seq(0, 1, by = 0.01)
n <- 3
z <- (n + 1)/2
Pr <- sapply(pi, function(p) sum(dbinom(z:n, size = n, prob = p)))
plot(pi, Pr, type = "l", xlab = "Probability of being correct (individual)",
    ylab = "Probability of being correct (group)")
```



When will bagging fail?

Now suppose that we consider the same simplified setup as before (independent classifiers), but each classifier has a misclassification rate:

$$\Pr(\hat{c}^b(x) = 2) = 0.6$$

Then by the same arguments,

$$\lim_{B \to \infty} \Pr(\bar{c}^b(x) = 2) \to 1$$

In other words, the bagged classifier is perfectly inaccurate as the number of bootstrapped data sets $B \to \infty$.

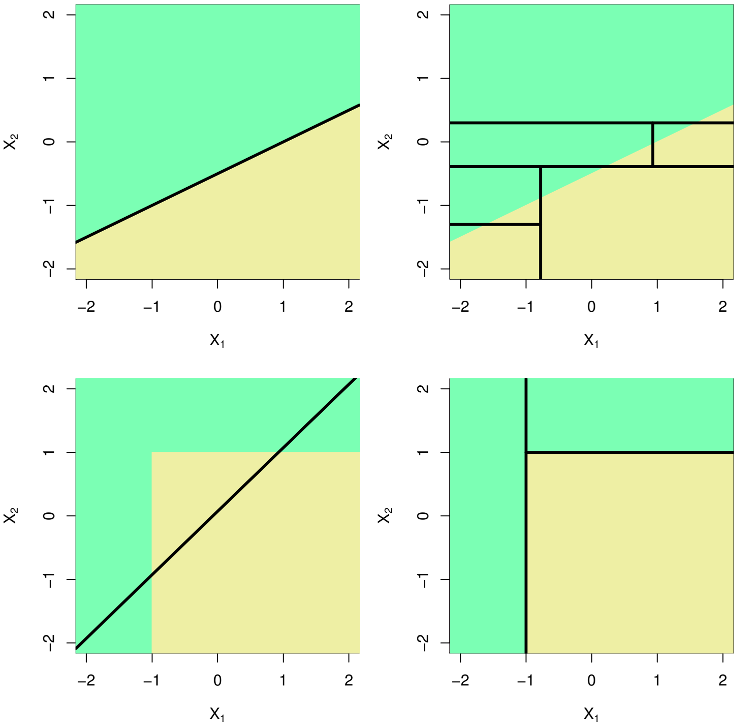
Again, the independence assumption doesn't hold with trees, but the take-away message is clear: **bagging a good classifier can improve predictive accuracy, but bagging a bad one can seriously degrade predictive accuracy**

Disadvantages

It is important to discuss some disadvantages of bagging:

- *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)

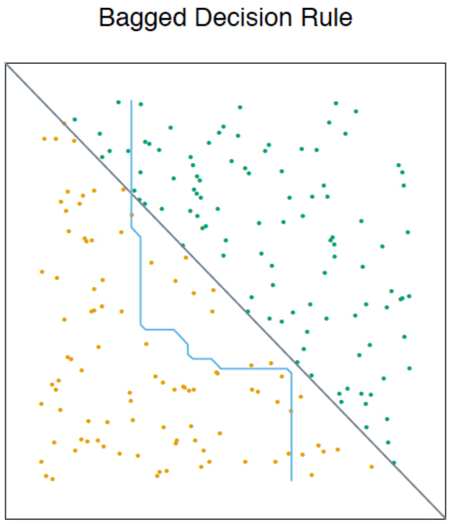
You can think of bagging as extending the space of models. We go from fitting a single tree to a large group of trees. Note that the final prediction rule cannot always be represented by a single tree. Sometimes, this enlargement of the model space isn't enough, and we would benefit from an even greater enlargement



- Top Row: A two-dimensional classification example in which the true decision boundary is linear, and is indicated by the shaded regions. A classical approach that assumes a linear boundary (left) will outperform a decision tree that performs splits parallel to the axes (right)
- Bottom Row: Here the true decision boundary is non-linear. Here a linear model is unable to capture the true decision boundary (left), whereas a decision tree is successful (right)

Example: limited model space

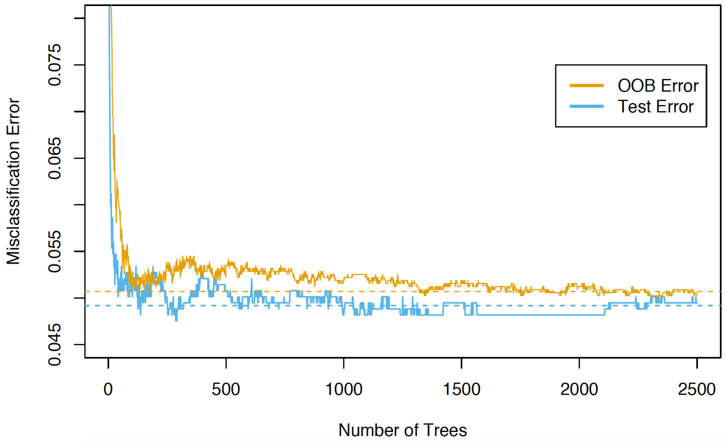
Example (from ESL page 288): bagging still can't really represent a diagonal decision rule



Source: ESL p. 288: Data with two features and two classes, separated by a linear boundary. The 100 data points shown have two features and two classes, separated by the gray linear boundary $x_1 + x_2 = 1$. Consider a classification rule with a single split along either x_1 or x_2 that produces the largest decrease in training misclassification error. Bagging doesn't help here. The decision boundary obtained from bagging over $B = 50$ bootstrap samples is shown by the blue curve in the left panel. The test error rate is 0.166

Out-of-bag error

- Each bagged tree makes use of $\approx 2/3$ of the original observations
- We can predict the response for the ith observation using each of the bagged trees in which that observation was OOB
- This yields $\approx B/3$ predictions for the ith observation, which we average (probability or consensus)
- This estimate is essentially the LOOCV error for bagging, if B is large



Source: ESL p. 592. OOB error computed on the spam training data, compared to the test error computed on the test set.

Example: Spam data

> Dear Google User, You have been selected as a winner for using our
> free services !!! Find attached email with more details.
> Its totally free and you won't be sorry !!! Congratulations !!!

Matt Brittin. CEO Google UK

Data description

- 4601 email messages sent to `George` at HP-Labs
- He labeled 1813 of these as `spam`, with the remainder being `good` email
- The goal is to build a customized spam filter for George: predict whether an e-mail message is `spam` or `good`
- Recorded for each email message is the relative frequency of certain key words (e.g. `business` , `address` , `free` , `George`) and certain characters: (, [, ! , $, #. Included as well are three different recordings of capitalized letters
- Publicly available dataset, available from the UC Irvine data repository. More details about the data can be found here
- For this problem not all errors are equal; we want to avoid filtering out good e-mail, while letting spam get through is not desirable but less serious in its consequences

Confusion matrix with K=2

The confusion matrix for the two-class problem: "positive class (or events)" and "negative class (or nonevents)". The table cells indicate number of the true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN)

	Observed	
Predicted	"positive class"	"negative class"
"positive class"	TP	FP
"negative class"	FN	TN
Total	P	N

One of these classes can be considered the event of interest or the **positive class** (e.g. `Yes` spam). One common way to think about performance is to consider false negatives and false positives

- the **Sensitivity** is the true positive rate

$$\text{sensitivity} = \frac{\#\,\text{Truly Yes predicted correctly}}{\#\,\text{Truly Yes}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- the **Specificity** is the rate of correctly predicted negatives, or true positive rate = 1 - false positive rate.

$$\text{specificity} = \frac{\#\,\text{Truly No predicted correctly}}{\#\,\text{Truly No}} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

*

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Types of errors

- false positive rate (= 1 - specificity). The fraction of negative examples that are classified as positive
- false negative rate (= 1 - sensitivity). The fraction of positive examples that are classified as negative.

We can change the two error rates by changing the threshold from 0.5 to some other value t in $[0; 1]$:

$$\widehat{\Pr}(Y = \text{Yes}|X = x) \geq \text{threshold}$$

and vary **threshold**.

- In our example, we will evaluate predictions in terms of accuracy and **specificity**, i.e.

$$\frac{\#\text{ correctly predicted emails}}{\#\text{ true emails}}$$

	george	you	your	hp	free	hpl	!	our	re	edu	remove
spam	0.00	2.26	1.38	0.02	0.52	0.01	0.51	0.51	0.13	0.01	0.28
email	1.27	1.27	0.44	0.90	0.07	0.43	0.11	0.18	0.42	0.29	0.01

Source: ESL, Table 1.1: Average percentage of words or characters in an email message equal to the indicated word or character. We have chosen the words and characters showing the largest difference between spam and email.

```r
# Load the data and split into training and test sets.
rm(list = ls())
spam <- read.csv("https://web.stanford.edu/~hastie/CASI_files/DATA/SPAM.csv",
    header = T)
spam$spam <- as.factor(ifelse(spam$spam == T, "spam", "email"))
train <- spam[!spam$testid, -2]
test <- spam[spam$testid, -2]
n <- nrow(train)
m <- nrow(test)
```

```r
# A function to calculate prediction accuracy and specificity
score <- function(phat, truth, name = "model") {
    ctable = table(truth = truth, yhat = (phat > 0.5))
    accuracy = sum(diag(ctable))/sum(ctable)
    specificity = ctable[1, 1]/sum(ctable[1, ])
    data.frame(model = name, accuracy = accuracy, specificity = specificity)
}
```

Null model

```r
phat.null <- rep(mean(train$spam == "spam"), m)
yhat.null <- ifelse(phat.null > 0.5, "spam", "email")

# confusion matrix
table(Predicted = yhat.null, True = test$spam)
```

```
         True
Predicted email spam
    email   941  595
```

```r
# score
score(phat.null, test$spam, name = "null")
```

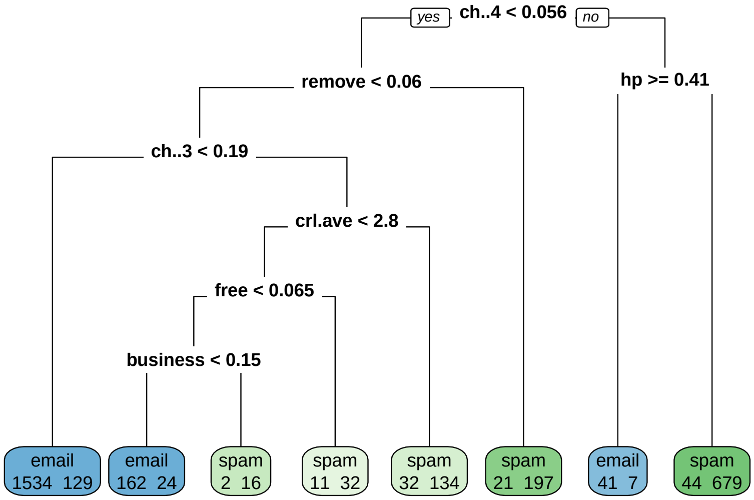
```
  model  accuracy specificity
1  null 0.6126302           1
```

Classification tree

```r
library(rpart)
fit.tree <- rpart(spam ~ ., train)
phat.tree <- predict(fit.tree, newdata = test)[, "spam"]
yhat.tree <- ifelse(phat.tree > 0.5, "spam", "email")


score(phat.tree, test$spam, name = "classification tree")
```

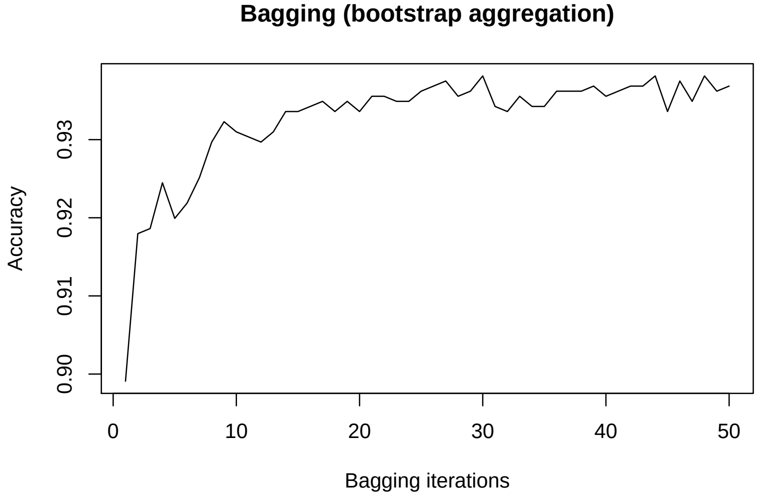
```
                model  accuracy specificity
1 classification tree 0.8977865   0.9330499
```

```r
library(rpart.plot)
rpart.plot(fit.tree, type = 0, extra = 1)
```

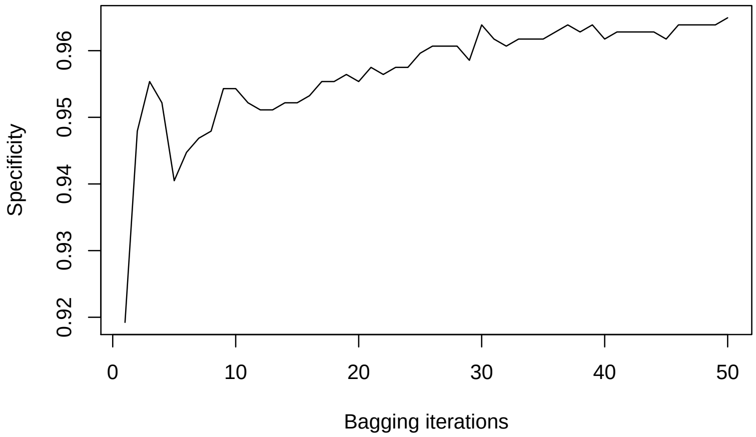


```
phat <- numeric(nrow(test))
acc <- NULL
spec <- NULL
B <- 50
for (b in 1:B) {
    use = sample(n, size = n, rep = T)
    fit.tree = rpart(spam ~ ., data = train[use, ], cp = 1e-04)
    phat = phat + predict(fit.tree, newdata = test, type = "prob")[,
        "spam"]
    yhat = ifelse(phat/b > 0.5, "spam", "email")
    acc = c(acc, mean(yhat == test$spam))
    spec = c(spec, score(phat/b, test$spam, name = "bagging")[3])
}
```

```
plot(1:B, acc, xlab = "Bagging iterations", ylab = "Accuracy",
    type = "l", main = "Bagging (bootstrap aggregation)")
```



```
plot(1:B, spec, xlab = "Bagging iterations", ylab = "Specificity",
    type = "l")
```



By function:

```
pnames <- setdiff(colnames(train), "spam")
fml <- as.formula(paste("spam", paste(pnames, collapse = " + "),
    sep = " ~ "))
fit1 <- rpart(fml, train)

phat1 <- predict(fit1, newdata = test)[, "spam"]
yhat1 <- ifelse(phat1 > 0.5, "spam", "email")
# confusion matrix
table(pred = yhat1, truth = test$spam)
```

```
       truth
pred    email spam
  email   878   94
  spam     63  501
```

```
# score
score(phat1, test$spam, name = "tree")
```

```
  model  accuracy specificity
1  tree 0.8977865   0.9330499
```

```
# Obtain B bootstrap samples
set.seed(123)
B <- 50
obs <- sapply(1:B, function(b) sample(1:n, size = n, replace = T))
# Train classification trees and return them in a list.
treelist <- lapply(1:B, function(b) rpart(fml, train[obs[, b],
    ]))

# predict.bag (probability)
```

```r
predict.bag <- function(treelist, newdata) {
    phats <- sapply(1:length(treelist), function(b) predict(treelist[[b]],
        newdata = newdata)[, "spam"])
    pbar <- rowMeans(phats)
}

phat2 <- predict.bag(treelist, newdata = test)
yhat2 <- ifelse(phat2 > 0.5, "spam", "email")
# confusion matrix
table(pred = yhat2, truth = test$spam)
```

```
       truth
pred    email spam
  email   890  100
  spam     51  495
```

```r
# score
score(phat2, test$spam, name = "bagging")
```

```
    model  accuracy specificity
1 bagging 0.9016927   0.9458023
```

Using caret:

```r
library(caret)
# Setting the random seed for replication
set.seed(123)

# setting up cross-validation
cvcontrol <- trainControl(method = "repeatedcv", number = 10,
    allowParallel = TRUE)

# a single classification Tree
train.tree <- train(spam ~ ., data = train, method = "ctree",
    tuneLength = 10)
train.tree
plot(train.tree)

# final tree: plot(train.tree$finalModel,
# main='Classification Tree for Spam Data')

# using treebag
train.bagg <- train(spam ~ ., data = train, method = "treebag",
    importance = TRUE)

train.bagg
```

```
# bagging within caret and use 10-fold CV
train.bagg2 <- train(spam ~ ., data = train, method = "treebag",
    trControl = trainControl(method = "cv", number = 10), )
train.bagg
```

Recap: bagging

Bagging is a technique in which we draw many bootstrap-sampled data sets from the original training data, train on each sampled data set individually, and then aggregate predictions at the end.

We applied bagging to classification trees. There were two strategies for aggregate the predictions:

- taking the class with the majority vote (the consensus strategy)
- averaging the estimated class probabilities and then voting (the probability strategy).

The former does not give good estimated class probabilities; the latter does and can sometimes even improve prediction accuracy.

Bagging works if the base classifier is not bad (in terms of its prediction error) to begin with. Bagging bad classifiers can degrade their performance even further. Bagging still can't represent some basic decision rules

Random forests

- Create even more variation in individual trees
- Bagging varies the **rows** of the training set (randomly draw observations)
- Random forests varies also the **columns** of the training set (randomly draw predictors)

Tuning parameter

- Before each split, select $m \leq p$ of the predictors at random as candidates for splitting
- m is a **tuning parameter**
- Typically $m = \sqrt{p}$ for classification and $m = p/3$ for regression
- $m = p$ gives Bagging as a special case

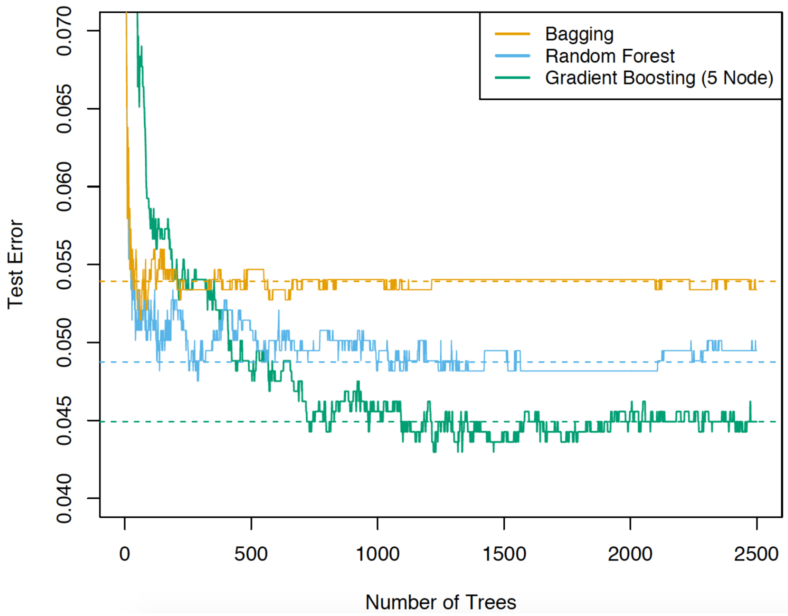
Why is random forests working?

- Random sampling of the predictors **decorrelates** the trees. This reduces the variance when we average the trees
- Recall that given a set of identical distributed (but not necessarily independent) variables $Z_1, \ldots, Z_B$ with pairwise correlation $\mathbb{C}\text{orr}(Z_j, Z_l) = \rho$, mean $\mathbb{E}(Z_j) = \mu$ and variance $\mathbb{V}\text{ar}(Z_j) = \sigma^2$, then

$$\mathbb{V}\text{ar}(\bar{Z}) = \rho\sigma^2 + \frac{(1-\rho)}{B}\sigma^2$$

- The idea in random forests is to improve the variance reduction of bagging by reducing the correlation ρ between the trees, without increasing the variance σ^2 too much
- $\rho = \frac{1}{\sigma^2}[\mathbb{E}(Z_i Z_j) - \mathbb{E}(Z_i)\mathbb{E}(Z_j)]$
- $\mathbb{E}(Z_i Z_j) = \rho\sigma^2 + \mu^2$ if $i \neq j$
- $\mathbb{E}(Z_i^2) = \sigma^2 + \mu^2$
- $\mathbb{E}[(\sum_{j=1}^{B} Z_j)^2] = \sum_{i=1}^{B}\sum_{j=1}^{B}\mathbb{E}(Z_i Z_j) = B\mathbb{E}(Z_i^2) + (B^2 - B)\mathbb{E}(Z_i Z_j)$
- $\mathbb{E}(\sum_{j=1}^{B} Z_j) = \sum_{j=1}^{B}\mathbb{E}(Z_j) = B\mu$

$$\begin{aligned}
\mathbb{V}\text{ar}(\bar{Z}) &= \frac{1}{B^2}\mathbb{V}\text{ar}(\sum_{j=1}^{B} Z_j) \\
&= \frac{1}{B^2}\{\mathbb{E}[(\sum_{j=1}^{B} Z_j)^2] - [\mathbb{E}(\sum_{j=1}^{B} Z_j)]^2\}
\end{aligned}$$



Source: ESL p. 589 Bagging and random forest applied to the spam data. For boosting, 5-node trees were used, and the number of trees were chosen by 10-fold cross-validation (2500 trees). Each "step" in the figure corresponds to a change in a single misclassification (in a test set of 1536).

randomForest()

The basic algorithm for a regression random forest can be generalized to the following:

```
Given training data set
Select number of trees to build (ntrees)
for i = 1 to ntrees do
|  Generate a bootstrap sample of the original data
|  Grow a regression tree to the bootstrapped data
|  for each split do
|  | Select m variables at random from all p variables
|  | Pick the best variable/split-point among the m
|  | Split the node into two child nodes
|  end
| Use typical tree model stopping criteria to determine when a tree is complete (but
end
```

```
randomForest(formula,
  data = ,
  ntree = ,       # default 500
  mtry = ,        # default sqrt(p) or p/3
  nodesize = ,    # default 1 or 5
  importance = ) # default: FALSE
```

- In bagging and random forests trees are grown large without pruning, only the minimum number of observations per node - `nodesize` - is fixed $= 1$ for classification and $= 5$ for regression
- Number of randomly selected predictors `mtry` $= \sqrt{p}$ for classification and $= p/3$ for regression. Note that `mtry` $= p$ is bagging

```
library(randomForest)
fit.rf <- randomForest(spam ~ ., data = train, ntree = B, importance = T)
phat.rf <- predict(fit.rf, newdata = test, type = "prob")[, "spam"]
yhat.rf <- predict(fit.rf, newdata = test)

table(pred = yhat.rf, truth = test$spam)
```

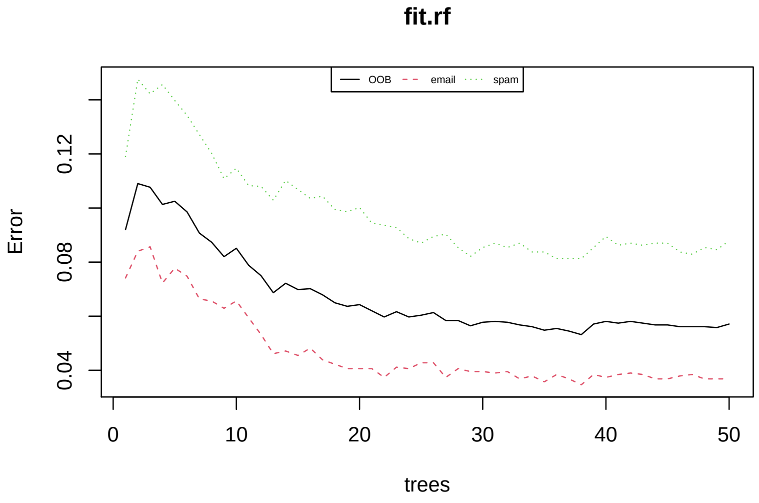
```
       truth
pred    email spam
  email   910   41
  spam     31  554
```

```
score(phat.rf, test$spam, name = "random forest")
```

```
          model accuracy specificity
1 random forest 0.953776   0.9691817
```

Similar to bagging, a natural benefit of the bootstrap resampling process is that random forests have an out-of-bag (OOB) sample that provides an efficient and reasonable approximation of the test error. This provides a built-in validation set without any extra work on your part, and you do not need to sacrifice any of your training data to use for validation. This makes identifying the number of trees required to stablize the error rate during tuning more efficient.

```
# out-of-bag estimate of error
plot(fit.rf)
```



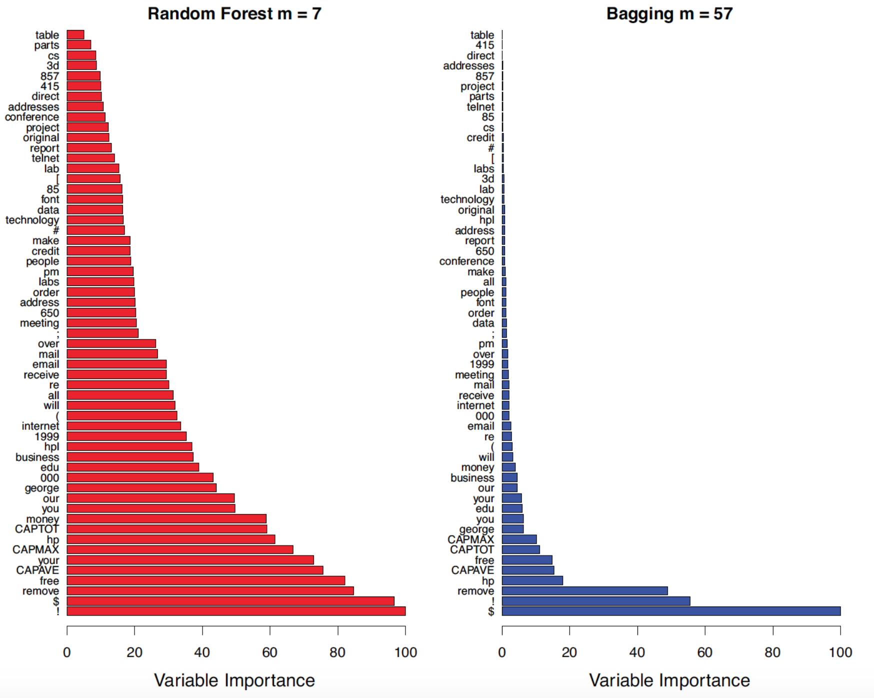
The black curve is the Out-of-Bag error rate. The red curve is the error rate for the `email` class, the green curve above is for `spam`.

Variable importance

- How might we get such a measure of variable importance from a random forest?
- The function `randomForests()` provides two different ways (argument `importance=TRUE`):

1. The first measure (`type=1`) is computed from permuting OOB (out-of-bag) data: For each tree, the prediction error on the out-of-bag portion of the data is recorded (error rate for classification, MSE for regression). Then the same is done after permuting each predictor variable (adding noise). The difference between the two are then averaged over all trees, and normalized by the standard deviation of the differences
2. The second measure (`type=2`) is the total decrease in node impurities from splitting on the variable, averaged over all trees. For classification, the node impurity is measured by the Gini index. For regression, it is measured by RSS

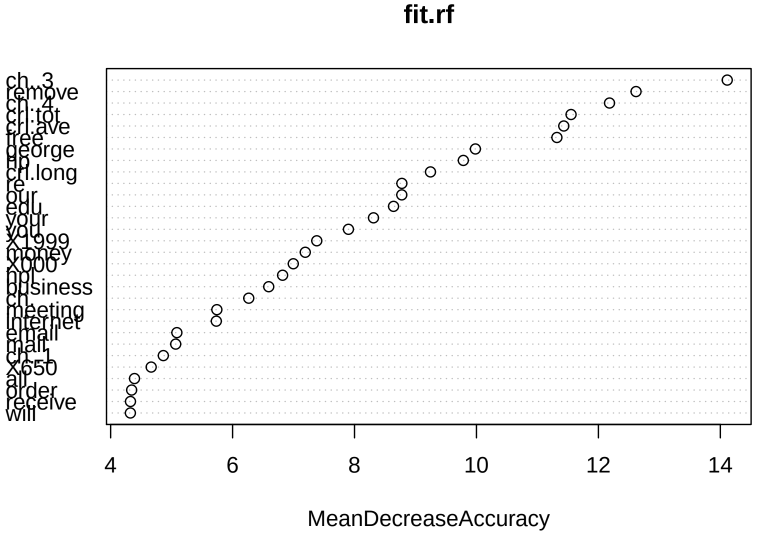
- So, the first compares the prediction of a variable with a random version of itself, while the second considers the error rates induced by splitting on a variable



Source: Efron and Hastie (2016) Computer Age Statistical Inference (CASI) p. 332.

Variable-importance plots for random forests fit to the spam data. On the left we have the $m = 7$ random forest; due to the split-variable randomization, it spreads the importance among the variables. On the right is the $m = 57$ random forest or bagging, which focuses on a smaller subset of the variables.

```r
varImpPlot(fit.rf, type = 1)
```



Random forest takeaways

- Bagging stabilizes decision trees and improves accuracy by reducing variance. Random forests further improve decision tree performance by de-correlating the individual trees in the bagging ensemble

- Random forests' variable importance measures can help you determine which variables are contributing the most strongly to your model

- Random forests are one of the better off-the-shelf methods. Strengths:

  - Easily incorporates features of different types (categorical and numeric)
  - Tolerance to irrelevant features
  - Some tolerance to correlated inputs
  - Good with big data
  - Handling of missing values

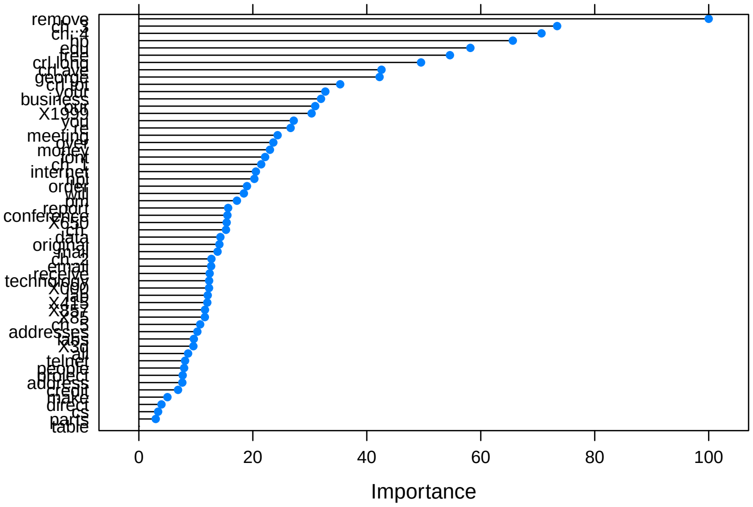
```
library("caret")
cv <- trainControl(
  method = "cv",
  number = 10,
  ## Estimate class probabilities
  classProbs = TRUE,
  ## Classification metrics
  summaryFunction = twoClassSummary
)
```

```
mtryGrid = data.frame(mtry = c(2, 29, 57))
rf <- train(spam ~ ., train, ntree = 50, method = "rf", tuneGrid = mtryGrid,
    localImp = TRUE, trControl = cv)
```

```
rf
```

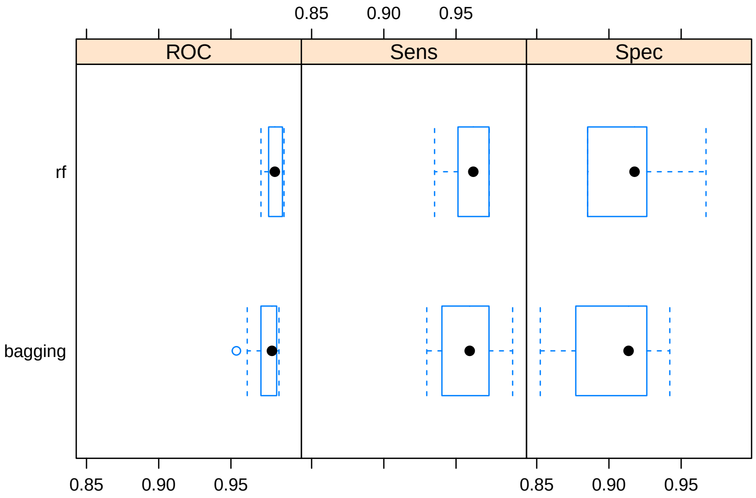
```
Random Forest

3065 samples
  57 predictor
   2 classes: 'email', 'spam'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 2758, 2758, 2759, 2759, 2758, 2759, ...
Resampling results across tuning parameters:

  mtry  ROC        Sens       Spec
   2    0.9801455  0.9723942  0.8842162
  29    0.9803666  0.9594036  0.9154247
  57    0.9770979  0.9599383  0.9121257

ROC was used to select the optimal model using the largest value.
The final value used for the model was mtry = 29.
```

```
plot(varImp(rf))
```

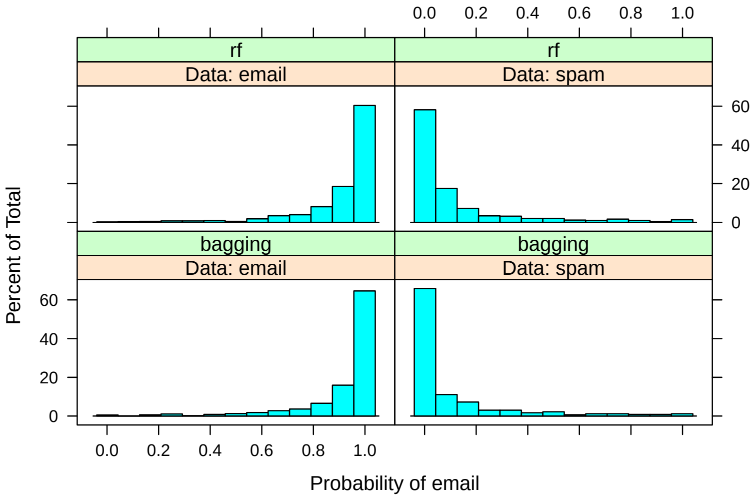


```
bagging <- train(spam ~ ., train, ntree = 50, method = "rf",
    tuneGrid = data.frame(mtry = 57), localImp = TRUE, trControl = cv)

models = list(bagging = bagging, rf = rf)
resamps <- resamples(models)
```

```
bwplot(resamps)
```



```
phats.all <- extractProb(models, testX = test, testY = test$spam)
phats <- subset(phats.all, dataType == "Test")
phats$model <- phats$object
plotClassProbs(phats)
```



Class Imbalance

Default data

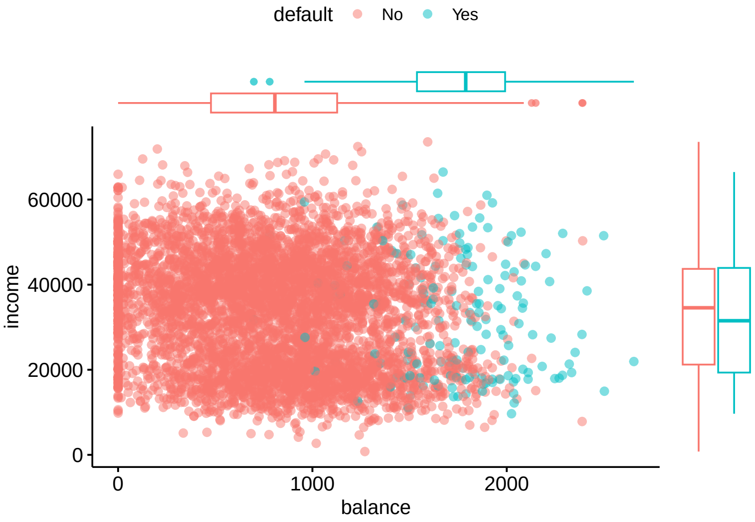
- A (simulated) data set containing information on 10000 customers
- We are interested in predicting whether an individual will `default` on his or her credit card payment, on the basis of annual income and monthly credit card balance
- `default` A factor with levels `No` and `Yes` indicating whether the customer defaulted on their debt
- `student` A factor with levels `No` and `Yes` indicating whether the customer is a student

- `balance` The average balance that the customer has remaining on their credit card after making their monthly payment

- `income` Annual income of customer

```
rm(list = ls())
library(ISLR)
n <- nrow(Default)/2
m <- n
set.seed(30)
istrain <- sample(c(rep(T, n), rep(F, length = m)))
train <- Default[istrain, ]
test <- Default[!istrain, ]
```

```
library(ggpubr)
ggscatterhist(train, x = "balance", y = "income", col = "default",
    alpha = 0.5, margin.plot = "boxplot")
```



```
# event of interest = 1
contrasts(train$default)
```

```
    Yes
No    0
Yes   1
```

```
# null model
fit0 <- glm(default ~ 1, train, family = "binomial")
phat0 <- predict(fit0, newdata = test, type = "response")
yhat0 <- ifelse(phat0 > 0.5, "Yes", "No")

# logistic model
fit <- glm(default ~ ., train, family = "binomial")
phat <- predict(fit, newdata = test, type = "response")
```

```r
yhat <- ifelse(phat > 0.5, "Yes", "No")
```

```r
# null model
table(predicted = yhat0, true = test$default)
```

```
         true
predicted   No  Yes
       No 4824  176
```

```r
mean(yhat0 == test$default)
```

```
[1] 0.9648
```

```r
# logistic model
table(predicted = yhat, true = test$default)
```

```
         true
predicted   No  Yes
      No  4800  119
      Yes   24   57
```

```r
mean(yhat == test$default)
```

```
[1] 0.9714
```

- With class predictions, a common summary method is to produce a confusion matrix which is a simple cross-tabulation between the observed and predicted classes
- Accuracy is the most obvious metric for characterizing the performance of models
- However, it suffers when there is a **class imbalance**; for the defaul data, 96.6% of the subjects have `No` default. Then we can expect 96.6% accuracy by predicting samples to be `No` default

Sensitivity and specificity

- One of these classes can be considered the event of interest or the **positive class** (e.g. `Yes` default)
- the **sensitivity** is the true positive rate

$$\text{sensitivity} = \frac{\#\,\text{Truly Yes predicted correctly}}{\#\,\text{Truly Yes}}$$

- the **specificity** is the rate of correctly predicted negatives, or 1 - false positive rate.

$$\text{specificity} = \frac{\#\,\text{Truly No predicted correctly}}{\#\,\text{Truly No}}$$

- Null model: sensitivity = 0/166=0%, specificity = 4834/4834 = 100%

- Logistic model: sensitivity = 59/166=35.5%, specificity = 4812/4834 = 99.5%
- Sensitivity and specificity are conditional statistics, e.g. if the custumer is truly Default, what is the probability that it is correctly predicted?

$$\Pr(P = \text{Yes}|Y = \text{Yes})$$

where Y denotes the true class and P denotes the prediction

- The question that one really wants to know is "if the custumer was predicted as Default, what is the probability that it is Default?", i.e.

$$\Pr(Y = \text{Yes}|P = \text{Yes})$$

- Bayes rule states that

$$\Pr(Y|P) = \frac{\Pr(Y)\Pr(P|Y)}{\Pr(P)}$$

- For sensitivity, its unconditional analog is called the **positive predictive value**

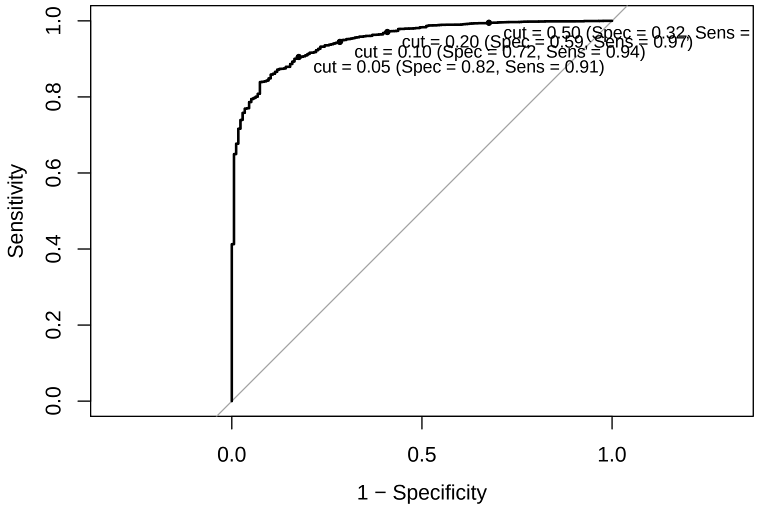
$$\text{PPV} = \frac{\text{sensitivity} \times \text{prevalance}}{(\text{sensitivity} \times \text{prevalance}) + [(1 - \text{specificity}) \times (1 - \text{prevalance})]}$$

- However, it may be difficult to provide a value for the prevalence

Changing the probability threshold

- For two classes, the 50% cutoff is customary; if the probability of default is >= 50%, they would be labelled as `Yes` default
- What happens when you change the cutoff?
- Increasing it makes it harder to be called `Yes` default. Then fewer predicted default events, higher specificity, lower sensitivity
- Decreasing the cutoff makes it easier to be called `Yes` default. Then more predicted events, lower specificity, higher sensitivity
- With two classes, the **Receiver Operating Characteristic** (ROC) curve can be used to estimate performance using a combination of sensitivity and specificity

```
Area under the curve: 0.949
```



The Receiver Operating Characteristic (ROC) Curve

- The ROC curve has some major advantages:
  - It can allow models to be optimized for performance before a definitive cutoff is determined
  - It is **robust** to class imbalances; no matter the event rate, it does a good job at characterizing model performance
- The ROC curve can be used to pick an optimal cutoff based on the trade-offs between the types of errors that can occur
- When there are two classes, it is advisable to focus on the **Area Under the Curve** (AUC) instead of sensitivity and specificity

Dealing with Class Imbalances

- In our data, only 3.3% of the subjects have default `Yes`
- This complicates the analysis since many models will overfit to the majority class
- There are two main strategies to deal with this:
  - **Cost-sensitive learning** where a higher cost is attached to the minority classes. In this way, the fitting process puts more emphasis on those samples
  - **Sampling procedures** that modify the rows of the data to re-balance the training set
- We will focus on sampling procedures

Class Imbalance Sampling

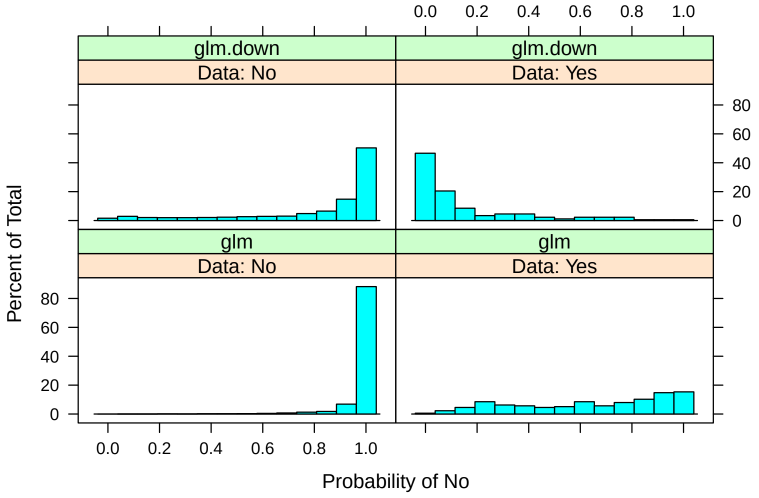
- There are a variety of methods for subsampling the data. Some exclude or replicate rows in the training set (**down-sampling** and **up-sampling**) while others try to synthesize new data points to balance the classes (**SMOTE** and **ROSE**)

- The simplest method for dealing with the problem is to down-sample the data to make the number of `Yes` and `No` the same
- While it seems like throwing away most of the data is a bad idea, it tends top produce less pathological distributions of the class probabilities and might improve the ROC curve
- It is critical that:
  - the sampling should be done **inside** of cross-validation. Otherwise, the performance estimates can be optimistic
  - these sampling methods take place on the training set and not the test set
- `caret` allows the user to specify down-sampling when using train so that it is conducted inside of cross-validation

```r
library(caret)
ctrl <- trainControl(method = "repeatedcv", number = 10, repeats = 2,
    classProbs = TRUE, summaryFunction = twoClassSummary)

# bagging
set.seed(123)
fit <- train(default ~ ., data = train, method = "glm", metric = "ROC",
    trControl = ctrl)

# bagging with downsampling
ctrl$sampling <- "down"
set.seed(123)
fit.down <- train(default ~ ., data = train, method = "glm",
    metric = "ROC", trControl = ctrl)
```

```r
phat <- predict(fit, test, type = "prob")[, "Yes"]
phat.down <- predict(fit.down, test, type = "prob")[, "Yes"]

yhat <- ifelse(phat > 0.5, "Yes", "No")
yhat.down <- ifelse(phat.down > 0.5, "Yes", "No")

phats <- data.frame(No = c(1 - phat, 1 - phat.down), Yes = c(phat,
    phat.down), obs = rep(test$default, 2), pred = c(yhat, yhat.down),
    model = c(rep("glm", m), rep("glm.down", m)), dataType = rep("Test",
        2 * m), object = c(rep("none", m), rep("down", m)))
```

```
plotClassProbs(phats)
```



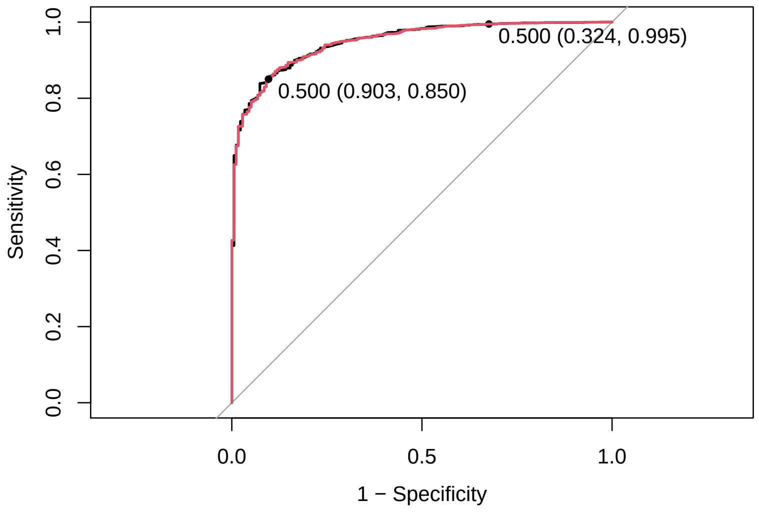
```
library(pROC)
# logistic regression
roc <- roc(response = test$default, predictor = phat, levels = c("Yes",
    "No"))
auc(roc)
```

```
Area under the curve: 0.949
```

```
# logistic regression with downsampling
roc.down <- roc(response = test$default, predictor = phat.down,
    levels = c("Yes", "No"))
auc(roc.down)
```

```
Area under the curve: 0.9484
```

```
plot(roc, legacy.axes = TRUE, print.thres = 0.5)
plot(roc.down, print.thres = 0.5, add = TRUE, col = 2)
```



Down-sampling and up-sampling

- Now we try down/up-sampling the training set before fitting
- We will see that cross-validation suggests up-sampling to be nearly perfect
- The reason that up-sampling appears to perform so well is that the observations in the `Yes` class are replicated and have a large potential to be in both the model building and hold-out sets
- This is related to the concept of **information leakage** which is where the hold-out set data are used (directly or indirectly) during the training process

```
library(caret)

# down-sampling
set.seed(123)
train_down <- downSample(x = train[, -1], y = train$default,
    yname = "default")
table(train_down$default)
```

```

 No Yes
157 157
```

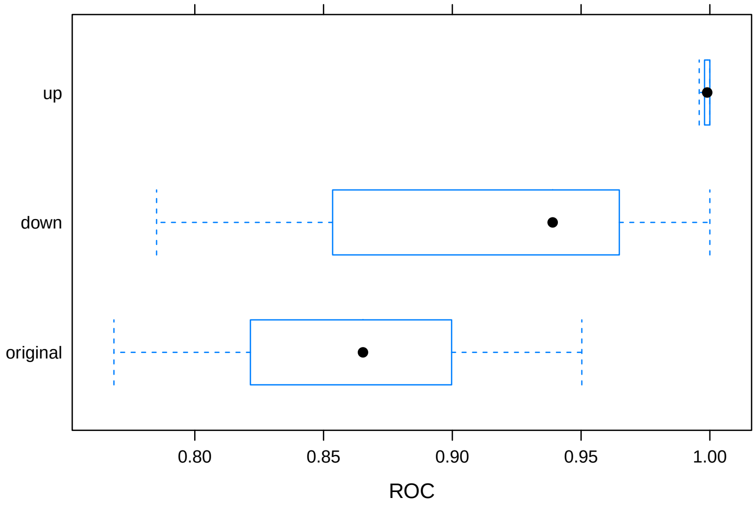
```
# up-sampling
set.seed(123)
train_up <- upSample(x = train[, -1], y = train$default, yname = "default")
table(train_up$default)
```

```

  No  Yes
4843 4843
```

```
ctrl <- trainControl(method = "cv", number = 10, classProbs = TRUE,
    summaryFunction = twoClassSummary)
set.seed(123)
fit <- train(default ~ ., data = train, method = "treebag", nbagg = 50,
    metric = "ROC", trControl = ctrl)
set.seed(123)
fit_down <- train(default ~ ., data = train_down, method = "treebag",
    nbagg = 50, metric = "ROC", trControl = ctrl)
set.seed(123)
fit_up <- train(default ~ ., data = train_up, method = "treebag",
    nbagg = 50, metric = "ROC", trControl = ctrl)
```

```
models <- list(original = fit, down = fit_down, up = fit_up)

res <- resamples(models)
```

```
bwplot(res, metric = "ROC")
```



```
phat <- predict(fit, test, type = "prob")[, "Yes"]
phat_down <- predict(fit_down, test, type = "prob")[, "Yes"]
phat_up <- predict(fit_up, test, type = "prob")[, "Yes"]

library(pROC)
roc(response = test$default, predictor = phat, levels = c("Yes",
    "No"))$auc
```

```
Area under the curve: 0.8666
```

```
roc(response = test$default, predictor = phat_down, levels = c("Yes",
    "No"))$auc
```

```
Area under the curve: 0.9146
```

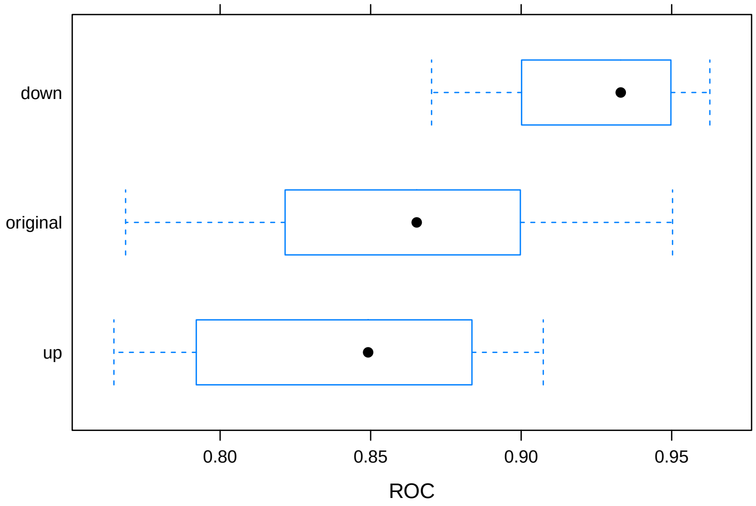
```
roc(response = test$default, predictor = phat_up, levels = c("Yes",
    "No"))$auc
```

```
Area under the curve: 0.8437
```

```
ctrl <- trainControl(method = "cv", number = 10, classProbs = TRUE,
    summaryFunction = twoClassSummary, sampling = "down")
set.seed(123)
fit_down <- train(default ~ ., data = train, method = "treebag",
    nbagg = 50, metric = "ROC", trControl = ctrl)
ctrl$sampling = "up"
set.seed(123)
fit_up <- train(default ~ ., data = train, method = "treebag",
    nbagg = 50, metric = "ROC", trControl = ctrl)
```

```
models <- list(original = fit, down = fit_down, up = fit_up)
res <- resamples(models)
```

```
bwplot(res, metric = "ROC")
```



```
phat <- predict(fit, test, type = "prob")[, "Yes"]
phat_down <- predict(fit_down, test, type = "prob")[, "Yes"]
phat_up <- predict(fit_up, test, type = "prob")[, "Yes"]

library(pROC)
roc(response = test$default, predictor = phat, levels = c("Yes",
    "No"))$auc
```

```
Area under the curve: 0.8666
```

```
roc(response = test$default, predictor = phat_down, levels = c("Yes",
    "No"))$auc
```

```
Area under the curve: 0.9274
```

```
roc(response = test$default, predictor = phat_up, levels = c("Yes",
    "No"))$auc
```

```
Area under the curve: 0.8497
```