# CHAPTER 4: COMPUTATION
## BOOTSTRAP AND RANDOM FOREST

## I. Overview

Chapter 3 is about "**Computation**". It is how we can work with computer, exploiting its remarkable power in iterations and conducting repetitive tasks which human beings often find burdensome to do. It is this capacity of computers in conducting repetitive tasks that enables applications of modern optimization algorithms, which underly many data analytics models. This capacity also provides powerful nonparametric statistical techniques, leading to developments of powerful computational statistical models that don't require analytic tractability. A particular invention that has played a critical role in many data analytic applications is the Bootstrap technique. Building on the idea of Bootstrap and its variants, Random Forest was also invented together with many more powerful ensemble learning methods.

## II. How Bootstrap Works

### II.1 Rationale and Formulation

There are multiple perspectives to look at Bootstrap. One perspective that has been well studied in the seminar book[1] of Efron and Tibshirani is to treat Bootstrap as a simulation of the "sampling process". As we know, sampling refers to the idea that we could draw samples again and again from the same population. Many statistical techniques make sense only when we consider the possibility of conducting sampling. For example, when we say the type 1 error in a hypothesis testing is 0.05, we mean that on average we may reject the null hypothesis 5 times even when it is true – if we conduct the same hypothesis testing 100 times (i.e., by repeating the sampling process 100 times, each time we calculate the test statistics and compare it with the critical value, and make a decision).

Of course, Bootstrap is not a real sampling process since no new data points are really collected. It is a simulated sampling process. Probably the idea of Bootstrap could be better demonstrated in Figure 4.1:

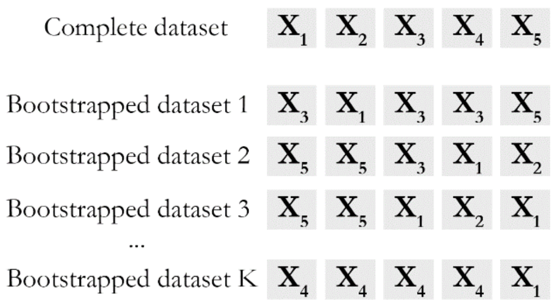
Complete dataset $X_1$ $X_2$ $X_3$ $X_4$ $X_5$
Bootstrapped dataset 1 $X_3$ $X_1$ $X_3$ $X_3$ $X_5$
Bootstrapped dataset 2 $X_5$ $X_5$ $X_3$ $X_1$ $X_2$
Bootstrapped dataset 3 $X_5$ $X_5$ $X_1$ $X_2$ $X_1$
...
Bootstrapped dataset K $X_4$ $X_4$ $X_4$ $X_4$ $X_1$

**Figure 4.1**: A demonstration of the Bootstrap process

As shown in Figure 4.1, a collected dataset has 5 samples. The 5 samples provide a representation of the underlying population. To mimic the

[1] *Bradley Efron and Robert J. Tibshirani. An Introduction to the Bootstrap. Chapman & Hall/CRC, 2993.*

sampling process that draws samples from the underlying population, Bootstrap suggests that we could resample the 5 samples to generate Bootstrapped datasets instead of really drawing samples from the underlying population. The idea seems to be simple but is very effective.

## II.2 Theory/Method

***The importance of the sampling process to classic statistics***: Many classic statistical theories are built on the sampling process. For example, let's consider the estimator of the mean of a normal population. Assume that we have a random variable $X$ that follows a normal distribution, $X \sim N(\mu, \sigma^2)$. For simplicity, let's assume that we have known the variance $\sigma^2$. So we want to estimate the mean $\mu$. What we need to do is to perform a sampling process, by randomly drawing a few samples from the distribution. Denote these samples as $x_1$, $x_2$, …, $x_n$. To estimate $\mu$, it seems natural to use the average of the samples, denoted as $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. Thus, we propose to use $\bar{x}$ as an estimator of $\mu$, i.e., $\hat{\mu} = \bar{x}$.

A question arises, how good is $\bar{x}$ to be an estimator of $\mu$?

To evaluate the uncertainty of the estimated $\mu$ by $\bar{x}$, in theory, we need to repeat the sampling process and the estimation again and again. If $\bar{x}$ is in theory a good estimator of $\mu$, then we should be able to repeatedly observe that $\bar{x}$ is numerically close to $\mu$ in the replications. Fortunately, when assuming that $X$ follows a normal distribution, we could derive that $\bar{x}$ is another normal distribution, $\bar{x} \sim N(\mu, \sigma^2/n)$. Thus, without really doing the physical experiments to repeat the sampling process and drawing many samples, we can answer the previous question. First, we know that $\bar{x}$ is an unbiased estimator as $E(\bar{x}) = \mu$. Also, we know that the larger the sample size, the better estimation of $\mu$ by $\bar{x}$, since the variance of the estimator is $\sigma^2/n$. Knowing the analytic form of $\bar{x}$ is the key here.

***Bootstrap – a computational remedy when the sampling process cannot be analytically articulated***: But how about we don't know what is the distribution of $X$? Then it would make the question difficult to answer as we don't know what is the analytic form of $\bar{x}$.

Bootstrap provides such a computational remedy that enables us to investigate the properties of literally any estimator by computationally mimicking the sampling process. For example, while the distribution of $X$ is unknown, we could follow the Bootstrap scheme illustrated in Figure 4.2 to evaluate the sampling distribution of $\bar{x}$.

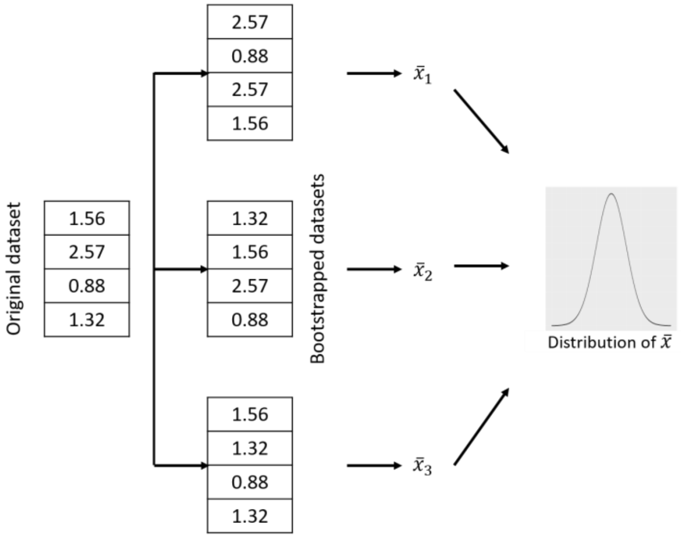

**Figure 4.2**: The (nonparametric) Bootstrap scheme for computationally evaluating the sampling distribution of $\bar{x}$

The Bootstrap scheme illustrated in Figure 4.2 is called nonparametric Bootstrap since no parametric information is used. This is not the only way we can conduct Bootstrap for studying the properties of any estimator in the sampling process. For example, a parametric Bootstrap scheme is illustrated in Figure 4.3 to perform the same study – to study the sampling distribution of $\bar{x}$. The only difference between the nonparametric Bootstrap scheme in Figure 4.2 and the parametric Bootstrap scheme in Figure 4.3 is that, when generating new samples, the nonparametric Bootstrap uses the original dataset while the parametric Bootstrap uses the fitted distribution model.

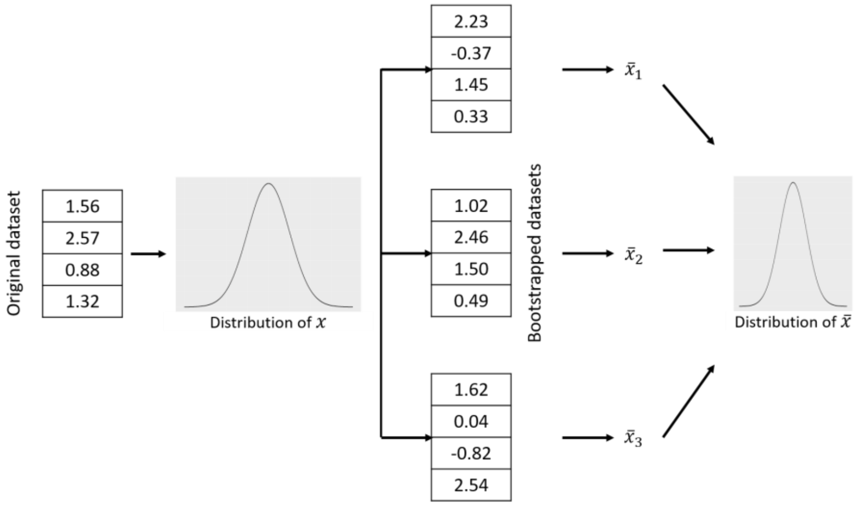

**Figure 4.3**: The (parametric) Bootstrap scheme for computationally evaluating the sampling distribution of $\bar{x}$

***Bootstrap for regression model***: How to evaluate the uncertainties of the regression parameters using Bootstrap?

In Chapter 2, we showed that by imposing the Gaussian assumptions on the error term of the regression model, we come to the recognition that the estimated regression parameters are also random variables, and the propagation of the uncertainty from the error term to the estimation of the regression parameters could be analytically articulated due to the benefit of the linear relationship assumed between predictors and outcome variable. In summary, the Gaussian assumption and the linear assumptions are critical for the analytic tractability.

Here, we introduce a more generic approach, based on the idea of Bootstrap, that could be applied on cases where we don't have to require those assumptions. Using Bootstrap to evaluate the linear regression model, we encounter a variety of options:

Option 1: we could simply resample the data points (i.e., the ($\boldsymbol{x}$,$y$) pairs) similarly as the nonparametric Bootstrap scheme. Then, for each sampled dataset, we can fit a regression model and obtain the fitted regression parameters. Suppose we repeat this sampling process 10,000 times, we could obtain 10,000 set of estimated regression parameters. These could be enough for us to evaluate the sampling distribution of the regression parameters and see if the parameters are significantly different from zero.

Option 2: we could simulate new samples of $X$ using the nonparametric Bootstrap method on the samples of $X$ only. Then, for the new samples of $X$, we draw samples of $Y$ using the fitted conditional distribution model $P(Y|X)$. This is a combination of the nonparametric and parametric Bootstrap methods to simulate $X$ and $Y$. Then, for each sampled dataset, we can fit a regression model and obtain the fitted regression parameters.

Option 3: we could fix the $X$, only sample for $Y$. In this way we implicitly assume that the uncertainty of the dataset mainly comes from $Y$. To sample $Y$, we draw samples using the fitted conditional distribution model $P(Y|X)$. In this way we could also generate many new datasets, such that we can generate many sets of fitted regression parameters.

The three options above are just some examples. There are other options that could be developed. As a matter of fact, as a more complicated model than simple parametric estimation in distribution fitting, how to conduct Bootstrap on regression models is a challenging problem that demands solutions from a variety of perspectives bearing different assumptions. Similarly, Bootstrap for other complex models such as time series models or decision tree models has demonstrated to be a challenging problem. This will be further discussed later in this Chapter when introducing the Random Forest.

## II.3 R Lab

In what follows we implement the Bootstrap scheme in statistical tasks such as parameter estimation, samples comparison, and regression models. First, let's load the AD dataset into the R workspace:

```
#### Dataset of Alzheimer's Disease
#### Objective: prediction of diagnosis
AD <- read.csv('AD_bl.csv', header = TRUE)
str(AD)
```

Let's pick up the variable `HippoNV`. The first statistical task we'd like to see is to estimate the mean of `HippoNV` in the population under study. Assuming that the variable `HippoNV` is distributed as a normal distribution, we could use the `fitdistr()` function from the R package "`MASS`" to estimate the parameters, mean and standard derivation, as shown in below:

```
require(MASS)

fit <- fitdistr(AD$HippoNV, densfun="normal")
```

The `fitdistr()` function returns the estimated parameters together with their standard derivation. Note that, here, the standard derivation of the estimated parameters is derived based on the statistical theory underlying this estimation, that builds on the assumption of normality of the variable.

```
fit

##       mean            sd
##   0.471662891   0.076455789
##  (0.003362522) (0.002377662)
```

To give a visual sense of this estimation, the R code in below shows the histogram of the variable `HippoNV` and the normal curve with the estimated parameters:

```
hist(AD$HippoNV, pch=20, breaks=25, prob=TRUE, main="")
curve(dnorm(x, fit$estimate[1], fit$estimate[2]), col="red", lwd=
2, add=T)
```

From Figure 4.4, it seems that the normality assumption is reasonable and the estimation of the parameters fits the empirical data well.

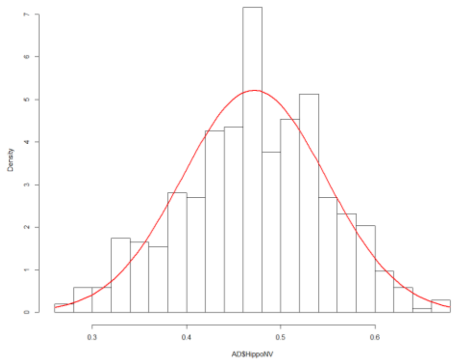

**Figure 4.4**: Histogram of `HippoNV` and its estimated normal curve

Now, let's focus on the question of how uncertain this estimation is. As we mentioned, the reason that the standard derivation of the estimated parameters could be provided by calling upon `fit()` is because the normality assumption is assumed. If we don't want to make this assumption, Bootstrap could be used in a nonparametrically way as shown in Figure 4.2. The following R codes implement the nonparametric Bootstrap for this parameter estimation problem:

```
# draw R bootstrap replicates
R <- 10000
# init location for bootstrap samples
bs_mean <- rep(NA, R)
# draw R bootstrap resamples and obtain the estimates
for (i in 1:R) {
  resam1 <- sample(AD$HippoNV, dim(AD)[1], replace = TRUE)
  fit <- fitdistr(resam1 , densfun="normal")
  bs_mean[i] <- fit$estimate[1]
}
```

Here, 10,000 replications are simulated by the Bootstrap method. The `(bs_mean` is a vector of 10,000 elements to record all the estimated mean parameter in these replications. These 10,000 estimated parameters could be taken as a set of samples. The following R code is used to compute the 95% CI of the estimated mean.

```
# sort the mean estimates to obtain bootstrap CI
bs_mean.sorted <- sort(bs_mean)
# 0.025th and 0.975th quantile gives equal-tail bootstrap CI
CI.bs <- c(bs_mean.sorted[round(0.025*R)], bs_mean.sorted[round(0.
975*R+1)])
CI.bs
```

It could be seen that this 95% CI is pretty much close to the 95% CI provided by theoretical result, showing the validity and efficacy of the Bootstrap method to evaluate the uncertainty of a statistical operation.

```
CI.bs

## [1] 0.4649877 0.4781062
```

The following R codes draws a hisogram of the `bs_mean` to give us some visual information about the Bootstrapped estimation of the mean.

```
## Plot the bootstrap distribution with CI
# First put data in data.frame for ggplot()
dat.bs_mean <- data.frame(bs_mean)

library(ggplot2)

## Warning: package 'ggplot2' was built under R version 3.3.3

p <- ggplot(dat.bs_mean, aes(x = bs_mean))
p <- p + geom_histogram(aes(y=..density..))
p <- p + geom_density(alpha=0.1, fill="white")
p <- p + geom_rug()
# vertical line at CI
p <- p + geom_vline(xintercept=CI.bs[1], colour="blue", linetype=
"longdash")
p <- p + geom_vline(xintercept=CI.bs[2], colour="blue", linetype=
"longdash")
p <- p + labs(title = "Bootstrap distribution of mean estimate of
 HippoNV")
print(p)
```

And the histogram is shown in Figure 4.5.

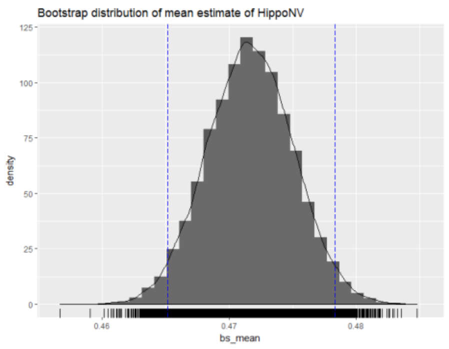

**Figure 4.5**: Histogram of the estimated mean parameter of `HippoNV` by Bootstrap with 10,000 replications

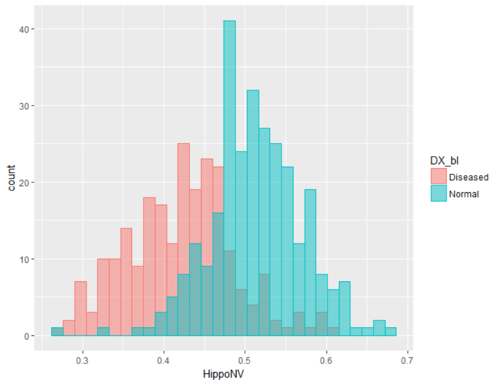

**Figure 4.6**: Histograms of `HippoNV` in the normal and diseased groups

While the estimation of the mean of `HippoNV` is a simple operation, in what follows we consider a relatively more complex statistical operation, comparison of the mean parameters of `HippoNV` across the two classes, normal and diseased. The following R code creates a temporary dataset for this purpose.

```
tempData <- data.frame(AD$HippoNV,AD$DX_bl)
names(tempData) = c("HippoNV","DX_bl")
tempData$DX_bl[which(tempData$DX_bl==0)] <- c("Normal")
tempData$DX_bl[which(tempData$DX_bl==1)] <- c("Diseased")
```

We then use ggplot to visualize the two distributions by comparing their histograms, which is shown in Figure 4.6.

```
p <- ggplot(tempData,aes(x = HippoNV, colour=DX_bl))
p <- p +  geom_histogram(aes(y = ..count.., fill=DX_bl), alpha=0.
5,position="identity")
print(p)
```

The following R code shows how the nonparametric Bootstrap method as shown in Figure 4.2 can be implemented here.

```
# draw R bootstrap replicates
R <- 10000
# init location for bootstrap samples
bs0_mean <- rep(NA, R)
bs1_mean <- rep(NA, R)
# draw R bootstrap resamples and obtain the estimates
for (i in 1:R) {
  resam0 <- sample(tempData$HippoNV[which(tempData$DX_bl=="Normal
")],
                   length(tempData$HippoNV[which(tempData$DX_bl==
"Normal")]),
                   replace = TRUE)
  fit0 <- fitdistr(resam0 , densfun="normal")
  bs0_mean[i] <- fit0$estimate[1]
  resam1 <- sample(tempData$HippoNV[which(tempData$DX_bl=="Diseas
ed")],
                   length(tempData$HippoNV[which(tempData$DX_bl==
"Diseased")]),
                   replace = TRUE)
  fit1 <- fitdistr(resam1 , densfun="normal")
  bs1_mean[i] <- fit1$estimate[1]
}

bs_meanDiff <- bs0_mean - bs1_mean

# sort the mean estimates to obtain bootstrap CI
bs_meanDiff.sorted <- sort(bs_meanDiff)
# 0.025th and 0.975th quantile gives equal-tail bootstrap CI
CI.bs <- c(bs_meanDiff.sorted[round(0.025*R)], bs_meanDiff.sorted
[round(0.975*R+1)])
CI.bs
```

Then, the 95% CI of the difference of the two mean parameters is:

```
CI.bs

## [1] 0.08066058 0.10230428
```

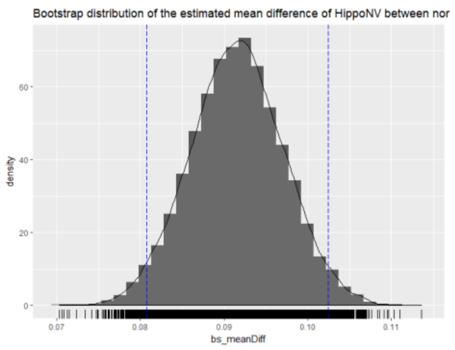

**Figure 4.7**: Histogram of the estimated mean difference of `HippoNV` in the two groups by Bootstrap with 10,000 replications

The following R codes draws a histogram of the `bs_meanDiff` to give us some visual information about the Bootstrapped estimation of the mean difference, which is shown in Figure 4.7.

```
## Plot the bootstrap distribution with CI
# First put data in data.frame for ggplot()
dat.bs_meanDiff <- data.frame(bs_meanDiff)

library(ggplot2)
p <- ggplot(dat.bs_meanDiff, aes(x = bs_meanDiff))
p <- p + geom_histogram(aes(y=..density..))
p <- p + geom_density(alpha=0.1, fill="white")
p <- p + geom_rug()
# vertical line at CI
p <- p + geom_vline(xintercept=CI.bs[1], colour="blue", linetype=
"longdash")
p <- p + geom_vline(xintercept=CI.bs[2], colour="blue", linetype=
"longdash")
p <- p + labs(title = "Bootstrap distribution of the estimated me
an difference of HippoNV between normal and diseased")
print(p)
```

Now let's implement the Bootstrap on the regression model, a more complicated statistical operation than the aforementioned two. First, we recall the use of `lm()` function to fit the regression model of `MMSCORE` using the demographics variables. Note that in this procedure, the standard derivation of the estimated regression parameters could be derived by theory (i.e., by assuming the error term is a normal distribution).

```
# Use Bootstrap for multiple regression model
tempData <- data.frame(AD$MMSCORE,AD$AGE, AD$PTGENDER, AD$PTEDUCA
T)
names(tempData) <- c("MMSCORE","AGE","PTGENDER","PTEDUCAT")
lm.AD_demo <- lm(MMSCORE ~ AGE + PTGENDER + PTEDUCAT, data = temp
Data)
summary(lm.AD_demo)
```

The fitted regression model is:

```
##
## Call:
## lm(formula = MMSCORE ~ AGE + PTGENDER + PTEDUCAT, data = tempD
ata)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.4290 -0.9766  0.5796  1.4252  3.4539
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 27.70377    1.11131  24.929  < 2e-16 ***
## AGE         -0.02453    0.01282  -1.913   0.0563 .
## PTGENDER    -0.43356    0.18740  -2.314   0.0211 *
## PTEDUCAT     0.17120    0.03432   4.988 8.35e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.062 on 513 degrees of freedom
## Multiple R-squared:  0.0612, Adjusted R-squared:  0.05571
## F-statistic: 11.15 on 3 and 513 DF,  p-value: 4.245e-07
```

Now, let's discard the assumption of normality of the error term, and use Bootstrap to induce perturbation into the data and see if the significance of the estimated parameters could resist this perturbation.

```
# draw R bootstrap replicates
R <- 10000
# init location for bootstrap samples
bs_lm.AD_demo <- matrix(NA, nrow = R, ncol = length(lm.AD_demo$co
efficients))
# draw R bootstrap resamples and obtain the estimates
for (i in 1:R) {
  resam_ID <- sample(c(1:dim(tempData)[1]), dim(tempData)[1], rep
lace = TRUE)
```

```
  resam_Data <- tempData[resam_ID,]
  bs.lm.AD_demo <- lm(MMSCORE ~ AGE + PTGENDER + PTEDUCAT, data =
 resam_Data)
  bs_lm.AD_demo[i,] <- bs.lm.AD_demo$coefficients
}
```

As `bs_lm.AD_demo` records all the estimated regression parameters in the 10,000 replications, here, for illustration purpose, we can take a look at the regression coefficient of the variable `AGE` by the following R code.

```
bs.AGE <- bs_lm.AD_demo[,2]
# sort the mean estimates of AGE to obtain bootstrap CI
bs.AGE.sorted <- sort(bs.AGE)

# 0.025th and 0.975th quantile gives equal-tail bootstrap CI
CI.bs <- c(bs.AGE.sorted[round(0.025*R)], bs.AGE.sorted[round(0.9
75*R+1)])
CI.bs
```

Then we can see the 95% CI of `AGE` is shown in below, which includes 0 in the range. This is consistent with the conclusion made in the aforementioned analysis which shows that the variable `AGE` is insignificant (i.e., `p-value=0.0563`) by t-test that is based on the normality assumption.

```
CI.bs

## [1] -0.053940482  0.005090523
```

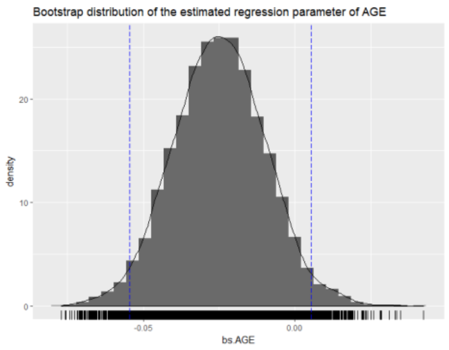

**Figure 4.8**: Histogram of the estimated regression parameter of AGE by Bootstrap with 10,000 replications

The following R codes draws a histogram of the Bootstrapped estimation of the regression parameter of AGE to give us some visual information about the Bootstrapped estimation, which is shown in Figure 4.8.

```
## Plot the bootstrap distribution with CI
# First put data in data.frame for ggplot()
dat.bs.AGE <- data.frame(bs.AGE.sorted)

library(ggplot2)
p <- ggplot(dat.bs.AGE, aes(x = bs.AGE))
p <- p + geom_histogram(aes(y=..density..))
p <- p + geom_density(alpha=0.1, fill="white")
p <- p + geom_rug()
# vertical line at CI
p <- p + geom_vline(xintercept=CI.bs[1], colour="blue", linetype=
"longdash")
p <- p + geom_vline(xintercept=CI.bs[2], colour="blue", linetype=
"longdash")
p <- p + labs(title = "Bootstrap distribution of the estimated re
gression parameter of AGE")
print(p)
```

Then we can see the 95% CI of PTEDUCAT as shown in below, which is between 0.1021189 and 0.2429209. This is consistent with the conclusion made in the aforementioned analysis which shows that the variable PTEDUCAT is significant (i.e., p-value=8.35e-07) by t-test that is based on the normality assumption.

```
bs.PTEDUCAT <- bs_lm.AD_demo[,4]
# sort the mean estimates of PTEDUCAT to obtain bootstrap CI
bs.PTEDUCAT.sorted <- sort(bs.PTEDUCAT)

# 0.025th and 0.975th quantile gives equal-tail bootstrap CI
CI.bs <- c(bs.PTEDUCAT.sorted[round(0.025*R)], bs.PTEDUCAT.sorted
[round(0.975*R+1)])
CI.bs

CI.bs

## [1] 0.1021189 0.2429209
```

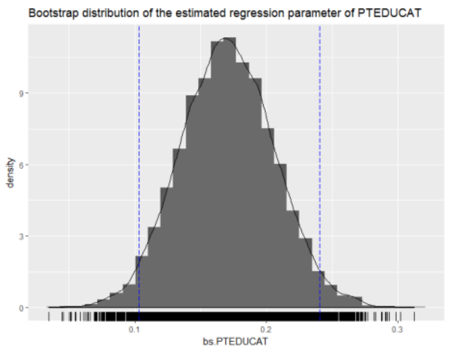

**Figure 4.9**: Histogram of the estimated regression parameter of `PTEDUCAT` by Bootstrap with 10,000 replications

The following R codes draws a histogram of the Bootstrapped estimation of the regression parameter of `PTEDUCAT`, which is shown in Figure 4.9.

```
## Plot the bootstrap distribution with CI
# First put data in data.frame for ggplot()
dat.bs.PTEDUCAT <- data.frame(bs.PTEDUCAT.sorted)

library(ggplot2)
p <- ggplot(dat.bs.PTEDUCAT, aes(x = bs.PTEDUCAT))
p <- p + geom_histogram(aes(y=..density..))
p <- p + geom_density(alpha=0.1, fill="white")
p <- p + geom_rug()
# vertical line at CI
p <- p + geom_vline(xintercept=CI.bs[1], colour="blue", linetype=
"longdash")
p <- p + geom_vline(xintercept=CI.bs[2], colour="blue", linetype=
"longdash")
p <- p + labs(title = "Bootstrap distribution of the estimated re
gression parameter of PTEDUCAT")
print(p)
```

## III. Random Forests

### III.1 Rationale and Formulation

Building on the decision tree model, a random forest consists of multiple tree models. There are two main sources for randomness. First, each tree is built on a randomly selected set of samples by applying Bootstrap on the original dataset. Second, in building a tree, specifically in splitting a node in the tree, a subset of features is randomly selected to choose the best split. Figure 4.10 shows this scheme of random forest.

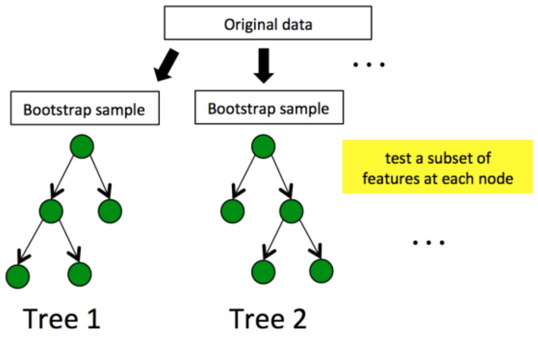

**Figure 4.10**: How random forest uses Bootstrap to grow trees

Random forest is a powerful machine learning method that has gained superior performances in many practical tasks, including many high-profiled data competitions over the past few years. It is a simple but effective mechanism to aggregate many simple models to tackle complex prediction task. Note that it is not necessary that in machine learning a random put-together of many simple models would lead to better performance than its constituting parts. Here we use the following example to show why the random forest, as a sum, is better than its parts.

The following R code generates a data set with two predictor variables and a class variable as the outcome variable. As shown in Figure 4.11, the two classes are separable by a linear boundary.

```
rm(list = ls(all = TRUE))
require(rpart)
require(dplyr)
require(ggplot2)
require(randomForest)
ndata <- 2000
X1 <- runif(ndata, min = 0, max = 1)
```

```
X2 <- runif(ndata, min = 0, max = 1)
data <- data.frame(X1, X2)
data <- data %>% mutate(X12 = 0.5 * (X1 - X2), Y = ifelse(X12 >=
0, 1, 0))
data <- data %>% select(-X12) %>% mutate(Y = as.factor(as.charact
er(Y)))
ggplot(data, aes(x = X1, y = X2, color = Y)) + geom_point() + lab
s(title = "Data points")
```

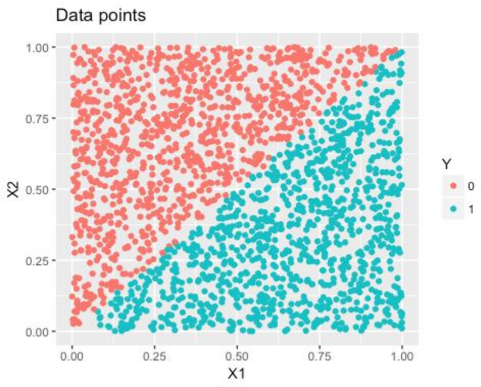

**Figure 4.11**: A linearly separable dataset with two predictors

Both the random forest and the decision tree are applied to the data. The classification boundaries the models can generate are shown in Figure 4.12 and Figure 4.13, for decision tree and random forest, respectively.

```
rf_model <- randomForest(Y ~ ., data = data)
tree_model <- rpart(Y ~ ., data = data)

pred_rf <- predict(rf_model, data, type = "prob")[, 1]
pred_tree <- predict(tree_model, data, type = "prob")[, 1]
data_pred <- data %>% mutate(pred_rf_class = ifelse(pred_rf < 0.5,
 0, 1)) %>%
    mutate(pred_rf_class = as.factor(as.character(pred_rf_class)))
 %>% mutate(pred_tree_class = ifelse(pred_tree <
    0.5, 0, 1)) %>% mutate(pred_tree_class = as.factor(as.charact
er(pred_tree_class)))
ggplot(data_pred, aes(x = X1, y = X2, color = pred_tree_class)) +
 geom_point() +
    labs(title = "Classification boundary from a single decision
tree")
```

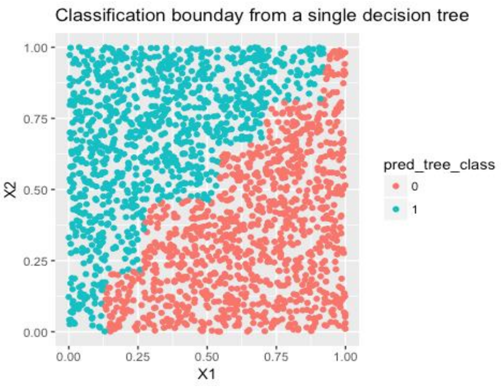

**Figure 4.12**: The decision boundary of one single decision tree

```
ggplot(data_pred, aes(x = X1, y = X2, color = pred_rf_class)) + g
eom_point() +
    labs(title = "Classification bounday from random forests")
```

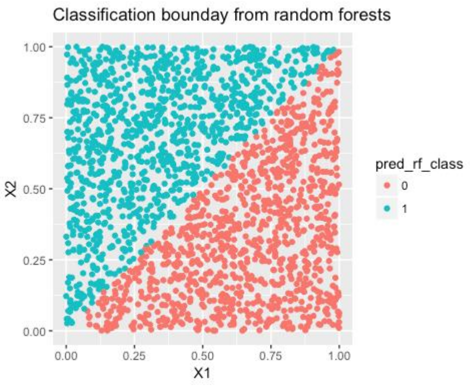

**Figure 4.13**: The decision boundary of a random forest

As we can see from Figure 4.12, the classification boundary generated by the decision tree model has difficult to approximate the linear boundary. There is an inherent limitation of a tree model to fit smooth boundaries due to its box-shaped nature resulting from its use of rules to segment the data space for making predictions. In contrast, the classification boundary of random forest is much smoother than the one of the decision tree, and is a better approximation of the linear classification boundary.

## III.2 Theory/Method

Pretty much like decision tree, the theoretical line of random forest follows the algorithmic modeling framework which is very different from the data modeling framework of the linear regression models. Thus, random forest is more of a systematically organized set of heuristics, rather than highly regulated algebraic operations derived from a mathematical characterization. Motivated by this recognition, we present the process of random forest using a simple example with the data shown in below.

| ID | X1 | X2 | Class |
|---|---|---|---|
| 1 | 1 | 1 | C0 |
| 2 | 1 | 0 | C1 |
| 3 | 0 | 1 | C1 |
| 4 | 0 | 0 | C0 |

For random forests with $m$ trees, each tree is built on a resampled dataset that consists of data instances randomly selected from the original data set, often with the same size as the original data set, **sampled with replacement**. As shown in Figure 4.14, the first resampled dataset includes data instances (represented by their IDs) {1,1,3,4} and is used for building the first tree. The second resampled dataset includes data instances (represented by their IDs) {2,3,4,4} and is used for building the second tree. And so on so forth, until the maximum number of trees is built.

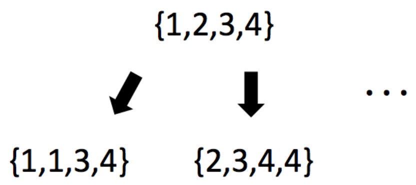

**Figure 4.14**: Bootstrap a dataset in random forest to build trees

To build the first tree, we begin with the root node that contains {1,1,3,4}. Then, we need to split the root node and reduce impurity. In the `randomForest` R package, the **Gini index** is used measure impurity.  The Gini index for a data set is defined as

$$Gini = \sum_{c=1}^{C} p_c(1 - p_c),$$

where $C$ is the number the classes in the dataset, and $p_c$ is the proportion of data instances that come from the class $c$.

The Gini index plays the same role as the entropy we have introduced in Chapter 2. Here, using the following R code, we plot the Gini index and entropy values versus the percentage of class 1 (for two-class problems) to see their similarity, as shown in Figure 4.15.

```
entropy <- function(p_v) {
    e <- 0
    for (p in p_v) {
        if (p == 0) {
            this_term <- 0
        } else {
            this_term <- -p * log2(p)
        }
        e <- e + this_term
    }
    return(e)
}
gini <- function(p_v) {
    e <- 0
    for (p in p_v) {
        if (p == 0) {
            this.term <- 0
        } else {
            this.term <- p * (1 - p)
        }
        e <- e + this.term
    }
    return(e)
}

entropy.v <- NULL
gini.v <- NULL
p.v <- seq(0, 1, by = 0.01)
for (p in p.v) {
    entropy.v <- c(entropy.v, (entropy(c(p, 1 - p))))
    gini.v <- c(gini.v, (gini(c(p, 1 - p))))
}
plot(p.v, gini.v, type = "l", ylim = c(0, 1), xlab = "percentage
of class 1",
    col = "red", ylab = "impurity measure", cex.lab = 1.5, cex.ax
is = 1.5, cex.main = 1.5,
    cex.sub = 1.5)
lines(p.v, entropy.v, col = "blue")
legend("topleft", legend = c("Entropy", "Gini index"), col = c("b
lue", "red"), lty = c(1, 1), cex = 0.8)
```

It can be seen in Figure 4.15 that the two impurity measures are highly correlated. Both reach minimum of zero when there is only one class in the dataset, and maximum when there are equal number of data instances for different classes. In practice, thus, they produce similar trees.

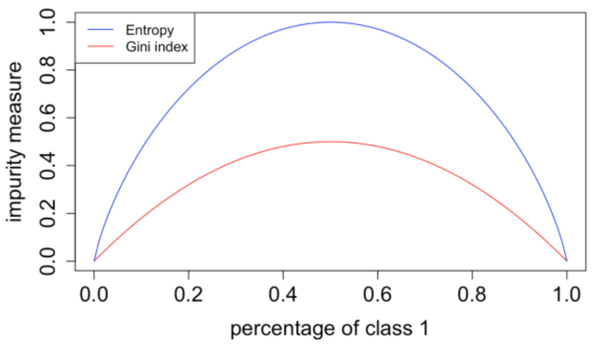

**Figure 4.15**: Gini index versus Entropy

Similar as the information gain, the **Gini gain** can be defined as

$$\nabla\, Gini = Gini - \ w_i Gini_i,$$

where $Gini$ is the Gini index at the node to be split; $w_i$ and $Gini_i$, are the proportion of samples and the Gini index at the $i^{th}$ children node, respectively.

Back to the example, the Gini index of the root node of the first tree is calculated as

$$\frac{3}{4} * \frac{1}{4} + \frac{1}{4} * \frac{3}{4} = 0.375.$$

The possible splitting rule candidates include four options: $X_1 = 0$, $X_2 = 0$, $X_1 = 1$ and $X_2 = 1$. Since both variables have two distinct values, both splitting rules $X_1 = 0$ and $X_1 = 1$ will produce the same children nodes, and both splitting rules $X_2 = 0$ and $X_2 = 1$ will produce the same children nodes. Therefore, we can reduce the possible splitting rule candidates to two: $X_1 = 0$ and $X_2 = 0$.

Further, as we mentioned earlier in this section, the second source of randomness in a random forest is to randomly select the variables for splitting a node. In general, for a data set with $p$ predictor variables, $\sqrt{p}$ variables are randomly selected for splitting. In our simple example, as there are two variables, we assume that $X_1$ is randomly selected for splitting the root node. Thus, $X_1 = 0$ is used for splitting the root node which generates the decision tree model as shown in Figure 4.16.

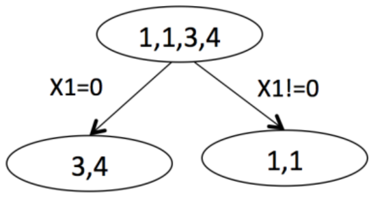

**Figure 4.16**: The decision tree with one split

The Gini gain can be calculated as

$$0.375 - 0.5 * 0 - 0.5 * 0.5 = 0.125.$$

Let's continue to grow the tree. Now, at the internal node containing data {3,4}, assume that $X_2$ is randomly selected. The node can be further split as shown in Figure 4.17.

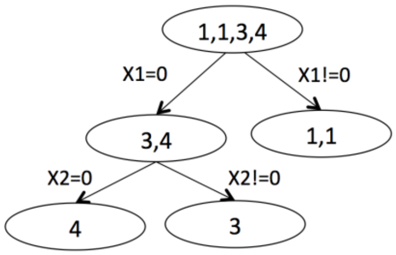

**Figure 4.17**: The decision tree with two splits

At this point, all nodes cannot be split further, and each leaf node can be labeled with the majority class of the node such that they become decision nodes. Thus, the final tree model is shown in Figure 4.18. Applying this decision tree to the 4 training data points, we can get the error rate as 25%.

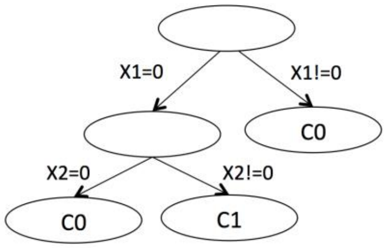

**Figure 4.18**: The final decision tree model with decision nodes

Similarly, the second, third, …, $m^{th}$ trees can be built. Usually, in random forest models, the pruning is not needed. Rather, we control the depth of the tree models to be created (i.e., use the parameter `nodesize` in the function `randomForest`).

To make a prediction for a data point, each tree makes a prediction for the data point, and the random forest model combines these predictions and selects the most popular prediction among all trees as the final prediction.

Note that, each tree in random forests can be weak classifier or even wrong model. But when in aggregation, the joint predictions become stronger. In what follows, we apply random forest on the toy example data with different number of trees. For each random forest model, we run the experiments 200 times and collect its overall performance using boxplots as shown in Figure 4.19.

```
require(dplyr)
require(ggplot2)
require(randomForest)
set.seed(1)
data <- rbind(c("0", "0", "C0"), c("1", "0", "C1"), c("0", "1", "
C1"), c("0",
    "0", "C0")) %>% as.data.frame()
colnames(data) <- c("X1", "X2", "Classs")

results <- NULL
for (i in c(1:9, (1:10) * 10)) {
    for (replicate in 1:200) {
        rf.model <- randomForest(Classs ~ ., data = data, ntree =
 i, keep.inbag = TRUE)
        pred.rf <- predict(rf.model, data, type = "class")
        err <- (length(which(pred.rf == data$Classs))/length(data
$Classs))
        results <- rbind(results, c(i, err))
    }
}
colnames(results) <- c("num_trees", "accuracy")
results <- as.data.frame(results) %>% mutate(num_trees = as.chara
cter(num_trees))
levels(results$num_trees) <- unique(results$num_trees)
results$num_trees <- factor(results$num_trees, unique(results$num
_trees))
ggplot() + geom_boxplot(data = results, aes(y = accuracy, x = num
_trees)) +
    geom_point(size = 3)
```

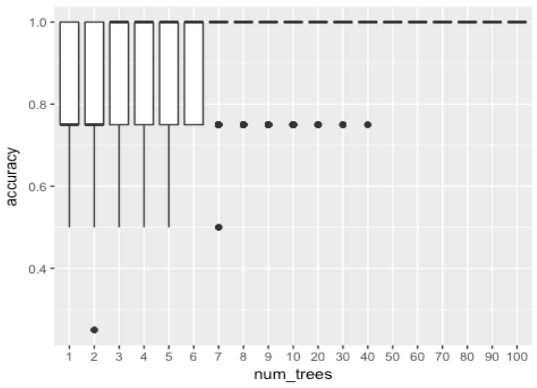

**Figure 4.19**: Accuracy versus number of trees in a random forest model

As shown in Figure 4.19, we can notice that when there are a small number of trees (e.g., smaller than 50), the performance is not stable. When the number of trees is greater than 50, the accuracy stabilizes. In practice, it is usually hard to say how many trees are best. Considerable amount of efforts of model tuning and selection is usually needed to make random forest works best on a dataset.

### III.3 R Lab

We apply both decision tree and random forests to the AD dataset. Half of the datasets are used for training and the other half for testing. This is run for 20 times, and the boxplots of the errors from decision tree and random forests are plotted in Figure 4.20 using the following R code.

```
library(rpart)
library(dplyr)
library(tidyr)
library(ggplot2)
require(randomForest)
set.seed(1)

theme_set(theme_gray(base_size = 15))
```

```
path <- "../../data/AD_bl.csv"
data <- read.csv(path, header = TRUE)

target_indx <- which(colnames(data) == "DX_bl")
data[, target_indx] <- as.factor(paste0("c", data[, target_indx]))
rm_indx <- which(colnames(data) %in% c("ID", "TOTAL13", "MMSCORE
"))
data <- data[, -rm_indx]

err.tree <- NULL
err.rf <- NULL
for (i in 1:20) {
    train.ix <- sample(nrow(data), floor(nrow(data)/2))
    tree <- rpart(DX_bl ~ ., data = data[train.ix, ])
    pred.test <- predict(tree, data[-train.ix, ], type = "class")
    err.tree <- c(err.tree, length(which(pred.test != data[-train.
ix, ]$DX_bl))/length(pred.test))

    rf <- randomForest(DX_bl ~ ., data = data[train.ix, ])
    pred.test <- predict(rf, data[-train.ix, ], type = "class")
    err.rf <- c(err.rf, length(which(pred.test != data[-train.ix,
 ]$DX_bl))/length(pred.test))
}
err.tree <- data.frame(err = err.tree, method = "tree")
err.rf <- data.frame(err = err.rf, method = "random_forests")

ggplot() + geom_boxplot(data = rbind(err.tree, err.rf), aes(y = e
rr, x = method)) +
    geom_point(size = 3)
```

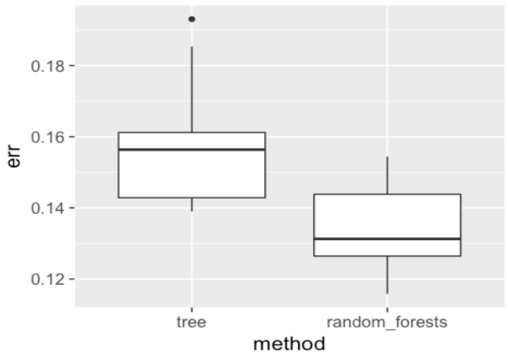

**Figure 4.20**: Performance of random forest versus tree model on the AD data

From Figure 4.20 we can see that the error rates of decision tree are higher than random forests. Now we investigate the impact of the number of trees and the number of features on the performance of random forest. First, let's consider the number of trees (i.e., use the parameter `ntree` in the function `randomForest`). For each number of trees, 20 runs are conducted, and the boxplots for each setting are shown in Figure 4.21.

```
library(rpart)
library(dplyr)
library(tidyr)
library(ggplot2)
require(randomForest)
set.seed(1)

theme_set(theme_gray(base_size = 15))

path <- "../../data/AD_bl.csv"
data <- read.csv(path, header = TRUE)

target_indx <- which(colnames(data) == "DX_bl")
data[, target_indx] <- as.factor(paste0("c", data[, target_indx]))
rm_indx <- which(colnames(data) %in% c("ID", "TOTAL13", "MMSCORE
"))
data <- data[, -rm_indx]

results <- NULL
for (itree in c(1:9, 10, 20, 50, 100, 200, 300, 400, 500, 600, 70
0)) {
    for (i in 1:20) {
        train.ix <- sample(nrow(data), floor(nrow(data)/2))
        rf <- randomForest(DX_bl ~ ., ntree = itree, data = data
[train.ix, ])
        pred.test <- predict(rf, data[-train.ix, ], type = "class
")
        this.err <- length(which(pred.test != data[-train.ix, ]$D
X_bl))/length(pred.test)
        results <- rbind(results, c(itree, this.err))
        # err.rf <- c(err.rf, length(which(pred.test !=
        # data[-train.ix,]$DX_bl))/length(pred.test) )
    }
}

colnames(results) <- c("num_trees", "error")
results <- as.data.frame(results) %>% mutate(num_trees = as.chara
cter(num_trees))
```

```r
levels(results$num_trees) <- unique(results$num_trees)
results$num_trees <- factor(results$num_trees, unique(results$num
_trees))
ggplot() + geom_boxplot(data = results, aes(y = error, x = num_tr
ees)) + geom_point(size = 3)
```

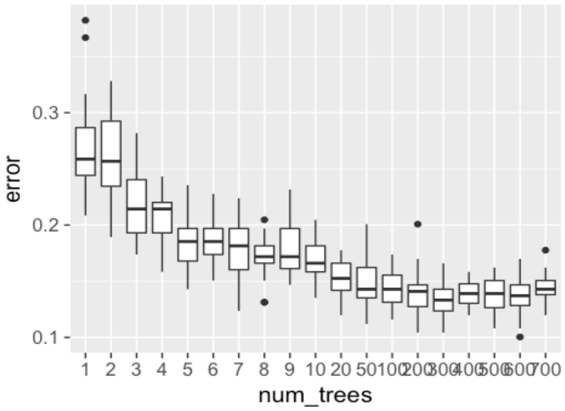

**Figure 4.21**: Error versus number of trees in a random forest model

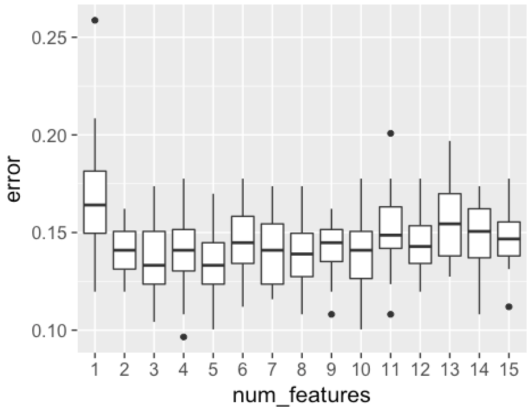

**Figure 4.22**: Error versus number of features in a random forest model

It can be seen in Figure 4.21 that, when the number of trees is small, particularly less than 10, the improvement on prediction performance of random forest is substantial with additional trees added. However, the error rates become stable after the number of trees reaches 100.

Next, let's consider the number of features (i.e., use the parameter `mtry` in the function `randomForest`). Here, 100 trees are used. For each number of features, 20 runs are conducted, and the boxplots for each setting are shown in Figure 4.22. It can be seen that the error rates are not significantly different when the number of features changes.

```
library(rpart)
library(dplyr)
library(tidyr)
library(ggplot2)
require(randomForest)
set.seed(1)
theme_set(theme_gray(base_size = 15))
path <- "../../data/AD_bl.csv"
data <- read.csv(path, header = TRUE)

target_indx <- which(colnames(data) == "DX_bl")
data[, target_indx] <- as.factor(paste0("c", data[, target_indx]))
rm_indx <- which(colnames(data) %in% c("ID", "TOTAL13", "MMSCORE
"))
data <- data[, -rm_indx]
nFea <- ncol(data) - 1
results <- NULL
for (iFeatures in 1:nFea) {
    for (i in 1:20) {
        train.ix <- sample(nrow(data), floor(nrow(data)/2))
        rf <- randomForest(DX_bl ~ ., mtry = iFeatures, ntree = 1
00, data = data[train.ix,
            ])
        pred.test <- predict(rf, data[-train.ix, ], type = "class
")
        this.err <- length(which(pred.test != data[-train.ix, ]$D
X_bl))/length(pred.test)
        results <- rbind(results, c(iFeatures, this.err))
        # err.rf <- c(err.rf, length(which(pred.test !=
        # data[-train.ix,]$DX_bl))/length(pred.test) )
    }
}

colnames(results) <- c("num_features", "error")
```

```
results <- as.data.frame(results) %>% mutate(num_features = as.ch
aracter(num_features))
levels(results$num_features) <- unique(results$num_features)
results$num_features <- factor(results$num_features, unique(resul
ts$num_features))
ggplot() + geom_boxplot(data = results, aes(y = error, x = num_fe
atures)) +
    geom_point(size = 3)
```

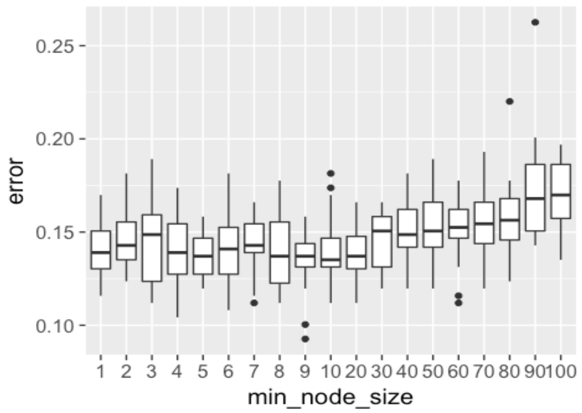

**Figure 4.23**: Error versus node size in a random forest model

Further, we experiment with the minimum node size (i.e., use the parameter `nodesize` in the function `randomForest`), that is, the minimum number of instances at a node. This is a parameter to control the depth of the trees. Again each setting is run 20 times and boxplots of their performances are shown in Figure 4.23.

```
library(dplyr)
library(tidyr)
library(ggplot2)
require(randomForest)
set.seed(1)

theme_set(theme_gray(base_size = 15))

path <- "../../data/AD_bl.csv"
```

```
data <- read.csv(path, header = TRUE)

target_indx <- which(colnames(data) == "DX_bl")
data[, target_indx] <- as.factor(paste0("c", data[, target_indx]))
rm_indx <- which(colnames(data) %in% c("ID", "TOTAL13", "MMSCORE
"))
data <- data[, -rm_indx]

results <- NULL
for (inodesize in c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50,
 60, 70, 80,
    90, 100)) {
    for (i in 1:20) {
        train.ix <- sample(nrow(data), floor(nrow(data)/2))
        rf <- randomForest(DX_bl ~ ., ntree = 100, nodesize = ino
desize, data = data[train.ix,
            ])
        pred.test <- predict(rf, data[-train.ix, ], type = "class
")
        this.err <- length(which(pred.test != data[-train.ix, ]$D
X_bl))/length(pred.test)
        results <- rbind(results, c(inodesize, this.err))
        # err.rf <- c(err.rf, length(which(pred.test !=
        # data[-train.ix,]$DX_bl))/length(pred.test) )
    }
}

colnames(results) <- c("min_node_size", "error")
results <- as.data.frame(results) %>% mutate(min_node_size = as.c
haracter(min_node_size))
levels(results$min_node_size) <- unique(results$min_node_size)
results$min_node_size <- factor(results$min_node_size, unique(res
ults$min_node_size))
ggplot() + geom_boxplot(data = results, aes(y = error, x = min_no
de_size)) +
    geom_point(size = 3)
```

It can be seen that, the error rates start to rise at minimum node size equal to 40. And the error rates are not substantially different when minimum node size less than 40. More importantly, this shows that a fully-grown tree, that is, minimum node size equal to 1, does not hurt the accuracy performance of random forests.

### III.4 Remarks

Random forest provides a great example to show when randomness should be consciously introduced into the model to boost its performance. This seems to be counterintuitive, as a model is supposed to characterize randomness and extract the constancy out of randomness. Actually, the randomness in the random forest, enabled by the use of Bootstrap to randomize choices of data instances and the use of random feature selection for building trees, is the key for its success. We provide an intuitive explanation that, why random forests work better than a single decision tree with the introduction of these randomness. Assuming that the trees in random forests are independent, and each tree has an accuracy of 0.6. For 100 trees, the probability of random forests to make the right prediction reaches as high as 0.97:

$$\sum_{k=51}^{100} C(n,k) * \ 0.6^{k} * 0.4^{100-k}.$$

Note that, the assumption of the independency between the trees in random forests is the key here. This does not hold in reality in a strict sense. However, the randomness added to each tree makes them less correlated.

## IV. Exercises

### *Data analysis*

1. Use `AGE` as the new outcome variable. Build a random forest model to predict it. Identify the final models you would select, evaluate the model, and compare it with the decision tree model.
2. Find two datasets from the UCI data repository or R datasets. Conduct a detailed analysis for both datasets using the random forest model. Also comment on the application of your model on the context of the dataset you have selected.
3. Pick up any dataset you have used, and randomly split the data into two halves. Use one half to build the random forest model. Test the model's prediction performance on the second half. Report what you have found, adjust your way of model building, and suggest a strategy to find the model you consider as the best.

***Programming***

4. Write your own R script to use Bootstrap to evaluate the significance of the regression parameters of logistic regression model. Compare your results with the output from `glm()`.
5. Write your own R script to implement the random forest algorithm. Use any dataset, compare the output from your script with the output from `randomForest()`.
6. Based on your script in 5, replace the decision tree model with logistic regression model, to generate a "random forest of logistic regression" model. Compare its performance with random forest on some datasets you have worked on.