8

Resampling Methods

8.1 INTRODUCTION

Resampling methods are a natural extension of simulation.[1] The analyst uses a computer to generate a large number of simulated samples, then analyzes and summarizes patterns in those samples. The key difference is that the analyst begins with the observed data instead of a theoretical probability distribution. Thus, in resampling methods, the researcher does not know or control the DGP. However, the goal of learning about the DGP remains.

Resampling methods begin with the assumption that there is some population DGP that remains unobserved, but that this DGP produced the one sample of observed data that a researcher has. The analyst then mimics the "in repeated samples" process that drives simulations by producing new "samples" of data that consist of different mixes of the cases in the original sample. That process is repeated many times to produce several new simulated "samples." The fundamental assumption is that all information about the DGP contained in the original sample of data is also contained in the distribution of these simulated samples. If so, then resampling from the one observed sample is equivalent to generating completely new random samples from the population DGP.[2]

Resampling methods can be parametric or nonparametric. In either type, but especially in the nonparametric case, they are useful because they allow the analyst to relax one or more assumptions associated with a statistical estimator. The standard errors of regression models, for example, typically rely on the central limit theorem or the asymptotic normality of ML estimates. What if there is good reason to suspect these assumptions are not valid? As Mooney and Duval (1993), point out, it is sometimes "better to draw conclusions about the characteristics of

[1]For book-length treatments of resampling methods, see Chernick and LaBudde (2011), Efron and Tibshirani (1993), Good (2005), or Mooney and Duval (1993).

[2]Another way to think about this is that if the sample of data you have in your hands is a reasonable representation of the population, then the distribution of parameter estimates produced from running a model on a series of resampled data sets will provide a good approximation of the distribution of that statistic in the population.

a population strictly from the sample at hand, rather than by making perhaps unrealistic assumptions about that population" (p. 1).

Within this context, the key question is how we can actually produce new simulated samples of data from the observed data that effectively mimic the "in repeated samples" framework. We discuss three common methods here: (1) permutation tests, in which the analyst "reshuffles" the observed data, (2) jackknifing, which involves iteratively dropping an observation and reestimating, and (3) bootstrapping, which uses repeated draws with replacement from the observed sample to create the simulated samples. Of these three, bootstrapping is the most versatile and widely used.

8.2 PERMUTATION AND RANDOMIZATION TESTS

Permutation tests are the oldest form of resampling methods, dating back to Ronald A. Fisher's work in the 1930s (e.g., Fisher, 1935; Pitman, 1937). They are typically used to test the null hypothesis that the effect of a treatment is zero. Rather than assuming a particular form for the null distribution, the analyst uses the observed data to create one. This is done by randomly shuffling the sample many times, creating new samples that "break" the relationship in the observed sample each time. Then, the statistic of interest is computed in each reshuffled sample. Finally, the estimate from the original sample is compared with the distribution of estimates from the reshuffled samples to evaluate how different the observed estimate is from random reshuffling. If every single reshuffling combination is computed, the procedure is called a permutation (or exact) test. Another option is to perform a "large" number of reshuffles, in which case it is called a randomization test. We briefly introduce these methods here. For more complete accounts, see Rosenbaum (2002), Good (2004), or Manly (2006).

Suppose you had a sample of individuals. One subset of them received a treatment while the remaining individuals did not. The actual sample of data records measures for every individual on a dependent variable of interest, Y, and whether that person received the treatment. Suppose you were interested in recording whether the means of Y differ for the two groups. A permutation test would reshuffle the data by keeping every individual's observed value of Y unchanged, but randomly assigning which individuals to record as having received the treatment and which to record as having not received the treatment. The difference in the means of Y between these two "groups" would be computed and saved. This process of reshuffling who was recorded as receiving the treatment or not would be done many times, with the difference in the means of Y being recorded each time. The actual observed difference in the means of Y for the original sample would then be compared with the distribution of these simulated differences in means that emerged from randomness. The objective is to evaluate whether the observed difference in means in the actual sample differs enough from the distribution of randomly generated ones for the researcher to conclude that the treatment has an impact on Y.

Permutation and randomization tests assume exchangeability, which means that the observed outcomes across individuals come from the same distribution regardless of the value(s) of the independent variable(s) (Kennedy, 1995). This is a weaker assumption than the iid assumption, which also includes the notion of independence. As we discuss later, resampling methods can be easily adapted to nonindependence between observations. Before getting to that point, we illustrate permutation and randomization tests with two basic examples.

8.2.1 A Basic Permutation Test

To develop an intuition for the logic of permutation tests, we start with a simple experiment using six observations. We create a true DGP in which there is a treatment effect: Cases in the treatment group (`observed.treatment = 1`) have larger values of the dependent variable (`observed.y`) than do those in the control group (`observed.treatment = 0`).

```
# Basic Permutation Test
library(combinat)
set.seed(98382)

case.labels <- letters[1:6] # ID labels for each case
observed.treatment <- c(1, 1, 1, 0, 0, 0) # Treatment assignment
# The dependent variable
observed.y <- rnorm(6, mean = observed.treatment*5, sd = 1)
# Put the data together
observed.data <- data.frame(case.labels, observed.treatment, observed.y)
observed.data
```

```
  case.labels observed.treatment observed.y
1           a                 1  5.2889932
2           b                 1  5.5227244
3           c                 1  5.7360698
4           d                 0 -0.6683198
5           e                 0  1.9418637
6           f                 0  0.9191380
```

The mean of the treatment group in this case is 5.52 compared with 0.73 for the control group (a difference of 4.79). We can conduct a difference-in-means test to arrive at the p value associated with observing this difference due to chance, which produces $t = 6.21$ and $p = 0.02$. However, this requires that we assume the difference we compute can be assumed to follow a t distribution. With a sample this small, it turns out that we can also compute the exact probability of observing this outcome with a permutation test.

The following function `p.test()` performs the permutation test. It takes the dependent variable (`y`), treatment variable (`treatment`), and case label variable (`labels`) as inputs. Then, it creates all of the possible combinations of

treatment assignment for those cases. In the observed data, treatment is assigned to cases a through f as 1,1,1,0,0,0, which means a, b, and c get the treatment and d, e, and f are in the control group. However, there could be many other possible assignments that could have happened, such as 0,1,1,0,1,0 or 1,0,1,0,0,1. In fact, there are exactly $\binom{6}{3} = 20$ possible combinations of three treatment and three control cases with a sample size of 6. The formula $\binom{N}{K}$ can be read as having a set of size N from which you choose K, or more simply as "N choose K." It is equal to $\frac{N!}{K! \times (N-K)!}$. In R, you could compute this by typing choose(6,3).[3]

The code that follows creates a function that will generate all 20 of these possible combinations. Note that it uses the combinat package.

```
p.test <- function(y, treatment, labels){ # Inputs: data, case labels
require(combinat) # Requires the combinat package

# This lists all possible combinations of treatment assignment
combinations <- unique(permn(treatment))
reshuffle.treatment <- matrix(unlist(combinations),
nrow = length(combinations), ncol = length(treatment), byrow = TRUE)

# Compute the difference-in-means for each combination
reshuffle.dm <- numeric(nrow(reshuffle.treatment))
for(i in 1:nrow(reshuffle.treatment)){
reshuffle.dm[i] <- mean(y[reshuffle.treatment[i, ] == 1]) -
mean(y[reshuffle.treatment[i, ] == 0])
}

# Return the difference-in-means for each combination
result <- cbind(reshuffle.treatment, reshuffle.dm)
colnames(result) <- c(as.character(labels), "DM")
return(result)
}
```

After listing each of these 20 combinations in the object reshuffle.treatment, the function computes the difference-in-means between the two groups for each combination. For example, for the combination 0,1,1,0,1, 0, it computes the difference-in-means between observations b, c, and e ("treatment") and observations a, d, and f ("control").

The result from applying this function to these data is shown below. Each row is a different combination of treatment assignment (note the first row is the assignment found in the observed data). The first six columns indicate whether

[3]Factorials can be computed with the factorial() command, though this command may produce incorrect answers due to rounding error that R encounters with very large integers. You can solve this problem by installing the gmp package and using its factorialZ() function.

each observation is in treatment (1) or control (0). The last column provides the difference-in-means of the dependent variable under that combination. The distribution of these means constitutes the null distribution of no difference between the treatment and control groups, which you can confirm by noting that the mean of this column is zero—the average difference between "treatment" and "control" groups when treatment is assigned randomly is zero.

```
      a b c d e f          DM
 [1,] 1 1 1 0 0 0   4.7850352
 [2,] 1 1 0 1 0 0   0.5154421
 [3,] 1 0 1 1 0 0   0.6576724
 [4,] 0 1 1 1 0 0   0.8134931
 [5,] 1 1 0 0 1 0   2.2555645
 [6,] 1 0 1 0 1 0   2.3977947
 [7,] 0 1 1 0 1 0   2.5536155
 [8,] 0 1 0 1 1 0  -1.7159776
 [9,] 1 0 0 1 1 0  -1.8717983
[10,] 0 0 1 1 1 0  -1.5737473
[11,] 0 0 1 1 0 1  -2.2555645
[12,] 0 1 0 1 0 1  -2.3977947
[13,] 0 1 1 0 0 1   1.8717983
[14,] 1 0 1 0 0 1   1.7159776
[15,] 1 0 0 1 0 1  -2.5536155
[16,] 1 1 0 0 0 1   1.5737473
[17,] 1 0 0 0 1 1  -0.8134931
[18,] 0 1 0 0 1 1  -0.6576724
[19,] 0 0 1 0 1 1  -0.5154421
[20,] 0 0 0 1 1 1  -4.7850352
```

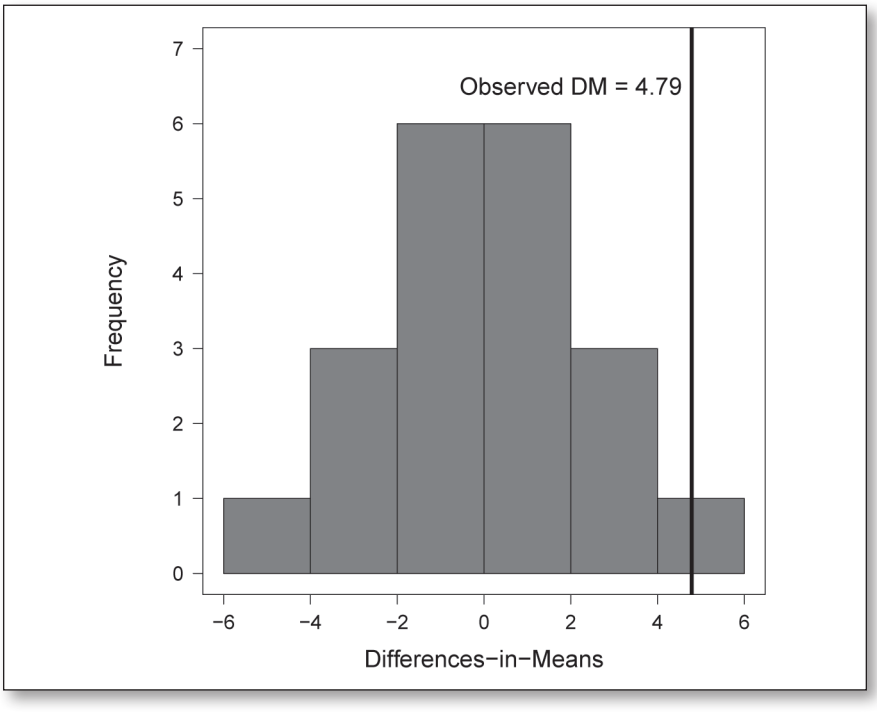
Notice that the observed difference, 4.79, is the largest of the 20 values. If the null hypothesis were true, we would expect to see this value in one out of every 20 samples, which corresponds to an exact p value of 0.05. Figure 8.1 shows a histogram of the results to reinforce this point. The observed difference-in-means (solid line) falls in the tail of the distribution—an indication that it is unlikely to appear due to random chance under the null hypothesis.

Finally, note that doing this procedure for larger data sets can become extremely complex. This small example with six observations produces 20 possible combinations of treatment assignment. However, a data set of 50 observations with 25 in the treatment group would produce $\binom{50}{25} = 126{,}410{,}606{,}437{,}752$ combinations! Clearly, this is more than we want to actually construct, which is why researchers move from permutation testing to randomization testing when the number of possible permutations becomes unwieldy.

8.2.2 Randomization Tests

A randomization test is a permutation test in which a large number of the possible permutations are assembled and analyzed instead of every one of them. We

Figure 8.1 Distribution of Differences-in-Means From All Permutations of the Null Distribution



illustrate how a randomization test works using data from the 1970s on the effect of the National Supported Work (NSW) job training programs on income (LaLonde, 1986). These data have been analyzed in several different ways and are part of a larger discussion of causal inference and experimental and observational studies in the social sciences (e.g., Dehejia & Wahba, 1999). We only use these data as an example; we do not make any causal claims from the results.

Our dependent variable in this example is the change in earnings between 1974 and 1978, which represent pre- and post-treatment measurement. Participants in the treatment group ($n = 185$) received job training during that period while those in the control group ($n = 429$) did not.[4] We evaluate whether the change in earnings was larger, on average, for those in the treatment group compared with those in the control group. The difference-in-means is $2,888.64, meaning that earnings for those in the program increased by nearly $3,000 more than earnings increased for those not in the program. Using a conventional t test, this estimate is statistically significantly different from zero ($p = 0.0001398$).

[4]The control group was constructed from survey data in the CPS.

We can also do the analysis with a randomization test.[5] The following function `r.test()` is very similar to `p.test()` from above. The difference is that while `p.test()` listed each combination of 1s and 0s to compute each permutation of treatment and control, `r.test()` computes 1,000 permutations by randomly drawing 1s and 0s with the `sample()` command. In other words, it takes a large random sample of all the possible permutations. This number can be set with the `reps` argument (the default is 1,000). As the number of randomly selected permutations gets larger, the randomization test gets closer and closer to an exact test.

```
# RT Function
r.test <- function(y, reps = 1000){ # Inputs: data, number of repetitions

# The sample command randomly draws a 1 or 0 for each observation
# The replicate command tells R to do this 'reps' times
# The result is a matrix in which each column is a new reshuffling
reshuffle.treatment <- replicate(reps, sample(0:1, length(y), replace = TRUE))

# Compute the difference-in-means for each reshuffling
reshuffle.dm <- numeric(reps)
for(i in 1:reps){
reshuffle.dm[i] <- diff(t.test(y ~ reshuffle.treatment[ , i])$estimate)
}
return(reshuffle.dm)
}
```

Figure 8.2 plots results from the function on the LaLonde (1986) data. The histogram shows the distribution of the average difference between the two groups when the treatment and control group labels were randomly shuffled. It is centered at zero with most of its values falling between –$1,000 and $1,000. The solid line at $2,888.64 shows the result from the observed sample. This observed difference is a very extreme value compared with the null distribution; in fact, none of the estimates computed in the randomization test are larger than the observed estimate.[6] Thus, it appears very unlikely that the observed difference between the two groups' change in earnings arose due to random chance, though, as we stated before, we do not wish to make any causal claims because there are issues with these data that we do not address here.[7]

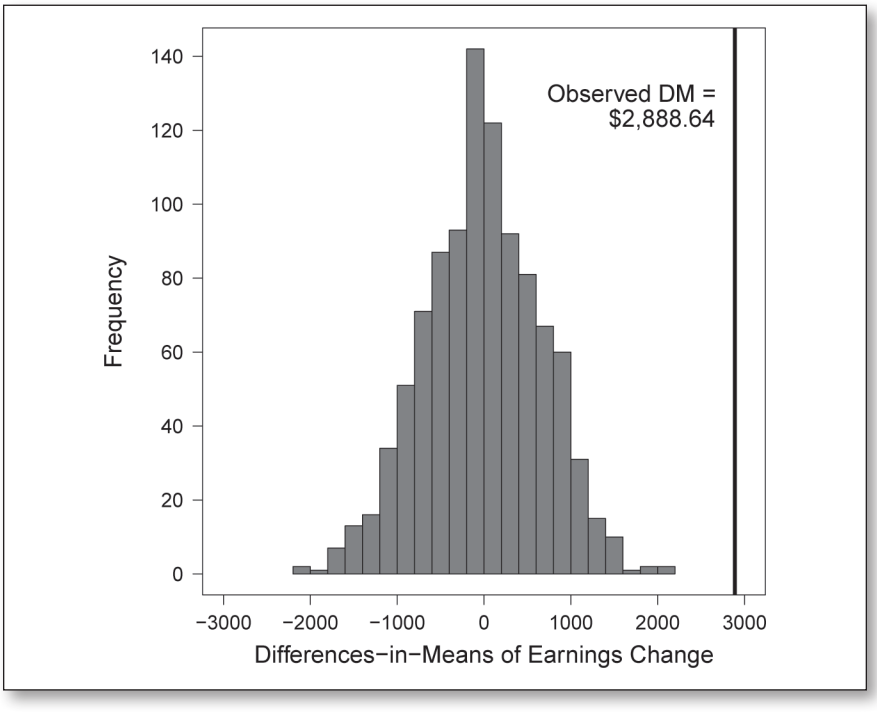
---

[5]Note that if we wanted to do a permutation test, the total number of combinations is an integer that starts with 5 and is followed by 161 additional digits—clearly too many to manage.

[6]We performed the randomization test again with `reps` set to 100,000. Even that many repetitions yielded no reshuffles that produced a larger difference than the observed value.

[7]Specifically, a more complete analysis would focus on whether the treatment and control groups are comparable to one another (see Dehejia & Wahba, 1999).

Figure 8.2 Distribution of Differences-in-Means From 1,000 Permutations of the Null Distribution in the LaLonde (1986) Job Training Data



8.2.3 Permutation/Randomization and Multiple Regression Models

These examples only show permutation and randomization testing in an experimental setting, which, in our view, is where they are best suited (see Keele, McConnaughy, & White, 2012). However, randomization experiments can also be extended to the regression framework. A key issue in using a randomization test with observational data and/or multiple regression models is the procedure for reshuffling. Kennedy (1995) reviews several possibilities, ultimately concluding that simply reshuffling the values of the independent variable of interest is sufficient (see also, Kennedy & Cade, 1996).[8] However, it is also recommended that when conducting inference, the analyst should store the *t* values in each reshuffle rather than the coefficient estimate on the variable of interest (Erikson, Pinto, & Rader, 2010). The reason for this is that while the DGP can naturally produce collinearity between independent variables, the reshuffling in a randomization test

[8]Other possibilities include reshuffling the values of the dependent variable or reshuffling the residuals across observations.

breaks that collinearity, which reduces the variability of the coefficient estimates that are computed on the permutation samples (recall the simulation from Chapter 5 on multicollinearity).

We advise caution when using randomization testing in a multiple regression model. The method was designed primarily to get a p value for a statistic of interest. This is different from estimation of the covariance matrix of a set of parameter estimates (and thus, their resulting standard errors). While we tend to focus on the standard errors of coefficients the most in social science applications, the estimates of covariance between two coefficients are often just as important. Parameter estimates will covary in statistical models—certainly in all of the models we described in Chapters 5 and 6—to the degree that any of the independent variables in those models are correlated with each other. To see this, we encourage you to return to the simulation on multicollinearity we presented in Chapter 5 and to plot the simulated estimates of the two slope coefficients operating on X_1 and X_2 for different levels of correlation between these two variables. You should see that as X_1 and X_2 become increasingly positively correlated, the estimates of their respective coefficients become increasingly negatively correlated. The covariance between model parameters is, thus, an important component of understanding a statistical model's uncertainty.

Fortunately, the full covariance matrix of a model's parameter estimates is a complete representation of model uncertainty, which can be used to compute a p value or a confidence interval. The problem is that randomization testing in a multiple regression setting—at least when the independent variables are reshuffled—does not allow for computing the covariance between coefficients. This is problematic because a covariance estimate is often needed for analysis of model results.[9] For instance, conducting joint F tests, calculating confidence intervals for the marginal effects of variables in interaction models, and simulating quantities of interest (which we will learn in Chapter 9) all *require* an estimate of covariance between coefficients of interest. This does not make randomization testing incorrect, but we feel it has limited applicability outside of the experimental setting or when simply comparing two groups, and readers should be aware of these limitations.

8.3 JACKKNIFING

The jackknife is another relatively old resampling method, dating back to the 1950s (e.g., Tukey, 1958). The goal is to create resampled data by iteratively deleting one observation from the data, computing the statistic of interest using the remaining data, then putting the observation back in and deleting another case.[10] This is done until each observation has been removed once. Thus, the

[9]This is less of a concern if the dependent variable or residual is reshuffled because doing so leaves the independent variable covariances intact. However, those approaches are less common in practice (see Erikson et al., 2010; Kennedy, 1995; Kennedy & Cade, 1996).

[10]This review draws primarily from Rizzo (2008).

number of resamples is equal to the size of the original sample. Formally, following Rizzo's (2008, p. 191) notation, if x is the observed random sample, the ith jackknife sample, $x_{(i)}$, is the subset of the sample that leaves out observation x_i: $x_{(i)} = (x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$. In most applications, jackknifing is used to estimate uncertainty. For a sample size n, the standard error of a statistic $\hat{\theta}$ is defined as

$$\sqrt{\frac{n-1}{n}\sum_{i=1}^{n}\left(\hat{\theta}_{(i)} - \bar{\hat{\theta}}_{(.)}\right)^2} \tag{8.1}$$

where $\bar{\hat{\theta}}_{(.)} = \frac{1}{n}\sum_{i=1}^{n}\hat{\theta}_{(i)}$, which is the mean of the estimates from each of the resamples (Rizzo, 2008, pp. 190–191).

Jackknifing can be accomplished in `R` by using the bracket notation. Recall from Chapter 3 that the minus sign can be used to specify all elements in a vector or matrix except a certain element. For example, the code `[-3]` would call all elements in a vector except for the third element. This can be used with the counter in a `for` loop to remove one observation at a time, as shown below.

```
v <- 1:5
for(i in 1:length(v)){
print(v[-i])
}
[1] 2 3 4 5
[1] 1 3 4 5
[1] 1 2 4 5
[1] 1 2 3 5
[1] 1 2 3 4
```

Imagine that the numbers printed out above are observation labels. In the first line, we could estimate the statistic of interest on all observations except #1; in the second line, we estimate it on all observations except #2, and so on.

8.3.1 An Example

We illustrate jackknifing with data on macroeconomic indicators for 14 countries during the period 1966–1990 that is available in the `Zelig` package (Imai et al., 2012). We specify an OLS model of the unemployment rate (`unem`) as a function of three independent variables: (1) GDP (`gdp`), (2) capital mobility (`capmob`), (3) trade (`trade`), and indicator variables for all countries except Austria (i.e., country fixed effects). An easy way to include indicator variables in `R` is to use the `factor()` function.

```
library(Zelig)
data(macro)

ols.macro <- lm(unem ~ gdp + capmob + trade + factor(country), data = macro)
summary(ols.macro)
```

```
Call:
lm(formula = unem ~ gdp + capmob + trade + factor(country), data = macro)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9811 -1.2594 -0.2674  0.9630  4.9581

Coefficients:

                                Estimate  Std. Error  t value   Pr(>|t|)
(Intercept)                     -5.84319     0.99674   -5.862   1.10e-08 ***
gdp                             -0.11016     0.04510   -2.443     0.0151 *
capmob                           0.81468     0.19156    4.253   2.75e-05 ***
trade                            0.14420     0.01138   12.669    < 2e-16 ***
factor(country)Belgium          -1.59865     0.66631   -2.399     0.0170 *
factor(country)Canada            6.75941     0.63342   10.671    < 2e-16 ***
factor(country)Denmark           4.31070     0.50798    8.486   7.14e-16 ***
factor(country)Finland           4.80987     0.56277    8.547   4.63e-16 ***
factor(country)France            6.90479     0.62070   11.124    < 2e-16 ***
factor(country)Italy             9.28969     0.60618   15.325    < 2e-16 ***
factor(country)Japan             5.45862     0.71636    7.620   2.66e-13 ***
factor(country)Netherlands      -1.45929     0.60332   -2.419     0.0161 *
factor(country)Norway           -2.75371     0.54409   -5.061   6.91e-07 ***
factor(country)Sweden            0.92533     0.52102    1.776     0.0766 .
factor(country)United Kingdom    5.60078     0.57706    9.706    < 2e-16 ***
factor(country)United States    10.06622     0.87617   11.489    < 2e-16 ***
factor(country)West Germany      3.36355     0.61641    5.457   9.49e-08 ***
---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 1.775 on 333 degrees of freedom
Multiple R-squared: 0.7137,   Adjusted R-squared: 0.6999
F-statistic: 51.88 on 16 and 333 DF, p-value: < 2.2e-16
```

This produces the conventional standard errors from the formula for OLS (OLS-SE). Let's compute jackknife standard errors (JSE) as a comparison using the following function, `jackknife()`, that we create. The function takes the names of the model and data objects as inputs. Then, it iterates through the data, removing one observation at a time, estimating the model again, and storing the results.[11] Once estimates have been computed with each observation removed, it applies the formula from Equation 8.1 to each coefficient.

```
jackknife <- function(model, data){ # Inputs: Model and data
n <- nrow(data) # Computes the sample size
```

[11]Notice the use of `formula()` to insert the model formula inside the function without needing to type it out.

```
# This creates an empty matrix to store the jackknife estimates
# Each row is one iteration and each column is a coefficient
jk.est <- matrix(NA, nrow = n, ncol = length(model$coef))

# The next step is to loop through the observations, removing
# one each time, estimating the model, then storing the results
for (i in 1:n){
jk.est[i, ] <- lm(formula(model), data = data[-i, ])$coef
}

# Empty vector for the SEs
jk.se <- numeric(ncol(jk.est))

# Loop through the coefficients, computing the SE for each one with
# the formula for JSE
for (i in 1:ncol(jk.est)){
jk.se[i] <- sqrt(n/(n-1)*sum((jk.est[ , i] - mean(jk.est[ , i]))^2))
}
return(jk.se)
}
```

We can now use this function to compute JSE. We also bind the coefficients, OLS-SE, and JSE together, round them to three digits, and print the results.

```
jk.macro <- jackknife(ols.macro, macro)
round(data.frame("Coefficients" = ols.macro$coef,
"OLS.SE" = sqrt(diag(vcov(ols.macro))), "JSE" = jk.macro), digits = 3)
```

	Coefficients	OLS.SE	JSE
(Intercept)	-5.843	0.997	1.089
gdp	-0.110	0.045	0.045
capmob	0.815	0.192	0.201
trade	0.144	0.011	0.013
factor(country)Belgium	-1.599	0.666	0.659
factor(country)Canada	6.759	0.633	0.615
factor(country)Denmark	4.311	0.508	0.459
factor(country)Finland	4.810	0.563	0.511
factor(country)France	6.905	0.621	0.721
factor(country)Italy	9.290	0.606	0.660
factor(country)Japan	5.459	0.716	0.699
factor(country)Netherlands	-1.459	0.603	0.567
factor(country)Norway	-2.754	0.544	0.490
factor(country)Sweden	0.925	0.521	0.475
factor(country)United Kingdom	5.601	0.577	0.673
factor(country)United States	10.066	0.876	0.898
factor(country)West Germany	3.364	0.616	0.626

In some cases, the OLS-SE and JSE are equal (to three digits) while in others they are not. Sometimes the OLS-SEs are larger and sometimes the JSEs are larger. It

does not appear that using JSE produces a substantial difference in statistical inference in this case. This leads to the question of when JSE might be beneficial to use over the conventional standard errors of a model. We examine such an example below.

8.3.2 An Application: Simulating Heteroskedasticity

Having shown the basic operation of the jackknife, our next task is to demonstrate when it might be useful. In this example, we take the simulation on heteroskedasticity from Chapter 5 and add the estimation of jackknifed standard errors to it. We compare this JSE with the conventional standard errors produced by OLS. Because jackknifing is an empirical, nonparametric means of estimating uncertainty that does not assume constant variance, we expect that the JSE will perform better than OLS-SE.

We described this simulation in more detail in Chapter 5. The most important points are that we generate the variance of the error term as a function of the independent variable, X. This produces larger error variance at higher values of X and smaller error variance at smaller values of X (see Figure 5.4). In this case, we set the sample size to 200 rather than 1,000 because the jackknife slows down the simulation by a considerable amount.[12]

The code for the simulation is given below. We use the same `jackknife()` function from the example above. Inside the `for` loop we generate the data, estimate the model, then store the estimate of β_1 (the coefficient on the independent variable) and its estimated OLS-SE and JSE.

```
# Heteroskedasticity
# CP Function
coverage <- function(b, se, true, level = .95, df = Inf){ # Estimate,
                                                          # standard error,
                                                          # true parameter,
                                                          # confidence level,
                                                          # and df
qtile <- level + (1 - level)/2 # Compute the proper quantile
lower.bound <- b - qt(qtile, df = df)*se # Lower bound
upper.bound <- b + qt(qtile, df = df)*se # Upper bound
# Is the true parameter in the confidence interval? (yes = 1)
true.in.ci <- ifelse(true >= lower.bound & true <= upper.bound, 1, 0)
cp <- mean(true.in.ci) # The coverage probability
mc.lower.bound <- cp - 1.96*sqrt((cp*(1 - cp))/length(b)) # Monte Carlo error
mc.upper.bound <- cp + 1.96*sqrt((cp*(1 - cp))/length(b))
return(list(coverage.probability = cp, # Return results
            true.in.ci = true.in.ci,
            ci = cbind(lower.bound, upper.bound),
            mc.eb = c(mc.lower.bound, mc.upper.bound)))
}
```

[12]This is because in each iteration of the simulation, the jackknife has to estimate the model n times.

```
set.seed(38586) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.jack <- matrix(NA, nrow = reps, ncol = 3) # Empty matrix to store the
                                                  # estimates
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 200 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X
gamma <- 1.5 # Heteroskedasticity parameter

for(i in 1:reps){ # Start the loop
Y <- b0 + b1*X + rnorm(n, 0, exp(X*gamma)) # Now the error variance is a
                                           # function of X plus random noise

model <- lm(Y ~ X) # Estimate OLS model
vcv <- vcov(model) # Variance-covariance matrix
par.est.jack[i, 1] <- model$coef[2] # Store the results
par.est.jack[i, 2] <- sqrt(diag(vcv)[2])
par.est.jack[i, 3] <- jackknife(model, data.frame(Y, X))[2]
cat("Just completed iteration", i, "\n")
} # End the loop
```

We then use the `coverage()` function from Chapter 5 to compare OLS-SE and JSE.

```
# OLS-SE
coverage(par.est.jack[ , 1], par.est.jack[ , 2], b1,
df = n - model$rank)$coverage.probability
[1] 0.883

# JSE
coverage(par.est.jack[ , 1], par.est.jack[ , 3], b1,
 df = n - model$rank)$coverage.probability
[1] 0.946
```

As we saw in Chapter 5, the OLS-SEs are too small in the presence of heteroskedasticity, on average, with a 95% confidence interval coverage probability of 0.88 in this example. In contrast, the JSEs are the correct size; the coverage probability is 0.946 with simulation error bounds that include 0.95: [0.932, 0.960]. Thus, we conclude that jackknifing is a better method for computing standard errors in the presence of heteroskedasticity than is the conventional method. Using the sample data to produce an estimate of uncertainty outperforms the OLS-SE, which assume constant variance.

8.3.3 Pros and Cons of Jackknifing

The jackknife certainly has advantages, such as its nonparametric nature that makes it robust to some assumption violations. Like other resampling methods, it

can also be adapted to many different data structures, such as clustered data, in which entire groups of observations (rather than just one observation) are dropped in each iteration. It is also good for detecting outliers and/or influential cases in the data. Indeed, its leave-one-out procedure is similar to Cook's D, which is often used to detect outliers in linear regression models (Cook, 1977).

However, there are also limitations to jackknifing. It does not perform as well if the statistic of interest does not change "smoothly" across repetitions. For example, jackknifing will underestimate the standard error of the median in many cases because the median is not a smooth statistic (see Rizzo, 2008, p. 194). In such cases, it is necessary to leave more than one observation out at a time (Efron & Tibshirani, 1993). Additionally, the jackknife can be problematic in small samples because the sample size dictates the number of repetitions/resamplings that are possible.[13] A key theme throughout this book has been that adding more repetitions increases the precision of an estimate, but the number of repetitions is capped at N for the jackknife. Fortunately, this limitation is not faced by the most versatile and perhaps most common resampling method: bootstrapping.

8.4 BOOTSTRAPPING

Bootstrapping was formally introduced by Efron (1979). It gets its name from the phrase "to pull oneself up by the bootstraps," which typically refers to a person improving her situation in life through her own efforts. Bootstrapping reflects this quality by getting the most information you can about a population DGP from the one sample of data you have. While there are several varieties of bootstrapping, at their core is a common process of simulating draws from the DGP using only the sample data.[14]

The bootstrap process usually unfolds according to the following steps. Denote a sample $S = \{x_1, x_2, x_3, \ldots, x_n\}$ of size n drawn from a population P. A statistic of interest, θ, that describes P can be estimated by calculating $\hat{\theta}$ from S. The sampling variability of $\hat{\theta}$ can then be calculated via the bootstrap in the following way:

1. Draw a sample of size n from *S with replacement* such that each element is selected with probability $\frac{1}{n}$. Denote this "bootstrap sample," S_{boot1}.
2. Calculate a new estimate of θ from S_{boot1}. Denote this bootstrap estimate $\hat{\theta}_1^*$.
3. Repeat Steps 1 and 2 J times, storing each $\hat{\theta}_j^*$ to create V, a vector of bootstrap estimates of the parameter of interest, θ.

[13]To see this illustrated, try computing JSE for a model from the Ehrlich (1973) crime data set used in Chapter 3, which has only 47 observations. The JSE differ considerably from the OLS-SE in that case.

[14]There are numerous treatments of bootstrapping that go into much greater detail than we can in this section. Interested readers should examine Efron and Tibshirani (1993), Good (2005), Mooney and Duval (1993), and chapter 7 of Rizzo (2008).

For a sufficiently large J, the vector V from Step 3 can be used to estimate a standard error and/or confidence interval for θ through several different methods.[15]

There are several key features of this process. First, the draws must be *independent*—each observation in S must have an equal chance of being selected. Additionally, the bootstrap sample drawn in Step 1 should be size n to take full advantage of the information in the sample (although some bootstrap methods draw smaller samples). Finally, resampling must be done *with replacement*. This means that in any given bootstrap sample, some individual observations might get selected more than once while others might not get selected at all. If replacement did not occur, every bootstrap sample of size n would be identical to each other and to the original sample.

To see this illustrated, consider the following sample of 10 countries. First, we sample without replacement. Notice that doing so produces the exact same sample of 10 countries (though in a different order). Computing any statistic of interest would be identical each time when sampling is done without replacement.

```
countries <- c("United States", "Canada", "Mexico", "England", "France",
"Spain", "Germany", "Italy", "China", "Japan")

sample(countries, replace = FALSE)
 [1] "France"  "Italy"  "China"  "Spain"  "United States"
 [6] "England" "Japan"  "Canada" "Mexico" "Germany"
```

Now, we sample with replacement three times. Notice that sometimes one or more countries do not make it into a particular sample and other times a particular country is repeated in a given sample. This produces the variation needed to compute measures of uncertainty in the statistic of interest.

```
sample(countries, replace = TRUE)
 [1] "United States" "Italy"         "Italy"    "China"         "Germany"
 [6] "Italy"         "England"       "Japan"    "Italy"         "Canada"

sample(countries, replace = TRUE)
 [1] "England"       "United States" "Germany"  "Mexico"        "Canada"
 [6] "England"       "United States" "England"  "United States" "Spain"

sample(countries, replace = TRUE)
 [1] "Canada"  "Italy"  "Spain"   "Canada"  "Italy"  "China"  "China"
 [8] "Japan"   "Spain"  "Germany"
```

Below we go through several different types of bootstrapping. The most important point to keep in mind is that bootstrapping is a lot like simulation. The difference is that instead of drawing multiple random samples from a theoretical

[15]We use a single parameter, θ, to illustrate how bootstrapping works, but θ could just as easily represent a set of parameters of a statistical model (e.g., the βs in a regression model).

DGP (as we do with simulation), we are drawing multiple random samples from the observed data.

8.4.1 Bootstrapping Basics

We start with a basic example of bootstrapping to compute the standard error of a mean, μ, and compare it with simulation. We first draw a sample from a normal distribution using `rnorm()`. Then, we compute the mean and standard error for that sample using the formula for the standard error of a mean: $\frac{\hat{\sigma}}{\sqrt{n}}$.

```
# Bootstrap a Mean
set.seed(34738)
n <- 500 # Sample size
n.boot <- 1000 # Number of bootstrap samples

b <- rnorm(n, 4, 5) # The sample, from a DGP of mean = 4, SD = 5
mean.b <- mean(b) # Sample mean
se.b <- sd(b)/sqrt(n) # SE of the mean
se.b
[1] 0.2065029
```

We can bootstrap the standard error of μ by resampling from the data many times (`n.boot` = 1,000 here) and computing the mean of each of those bootstrapped samples. We do this in a `for` loop with the code below. The object `ind` is the key component. That object is a sample of observation numbers drawn with replacement. We insert that object into the brackets in the next line—`b[ind]`—to reference those observation numbers in the sample.

```
boot.b <- numeric(n.boot) # Vector for the bootstrap samples
for(i in 1:n.boot){ # Start the loop
# Observation indicators for the bootstrap sample
ind <- sample(1:n, replace = TRUE)
boot.b[i] <- mean(b[ind]) # Compute the mean of the bootstrap sample
} # End the loop
```

We can also draw 1,000 samples from the true DGP to compare with the bootstrap estimates. Remember that this is what bootstrapping is trying to mimic, so the two procedures should look very similar.

```
# Random draws from the DGP
dgp.b <- numeric(n.boot)
for(i in 1:n.boot){
dgp.b[i] <- mean(rnorm(n, 4, 5))
}
```

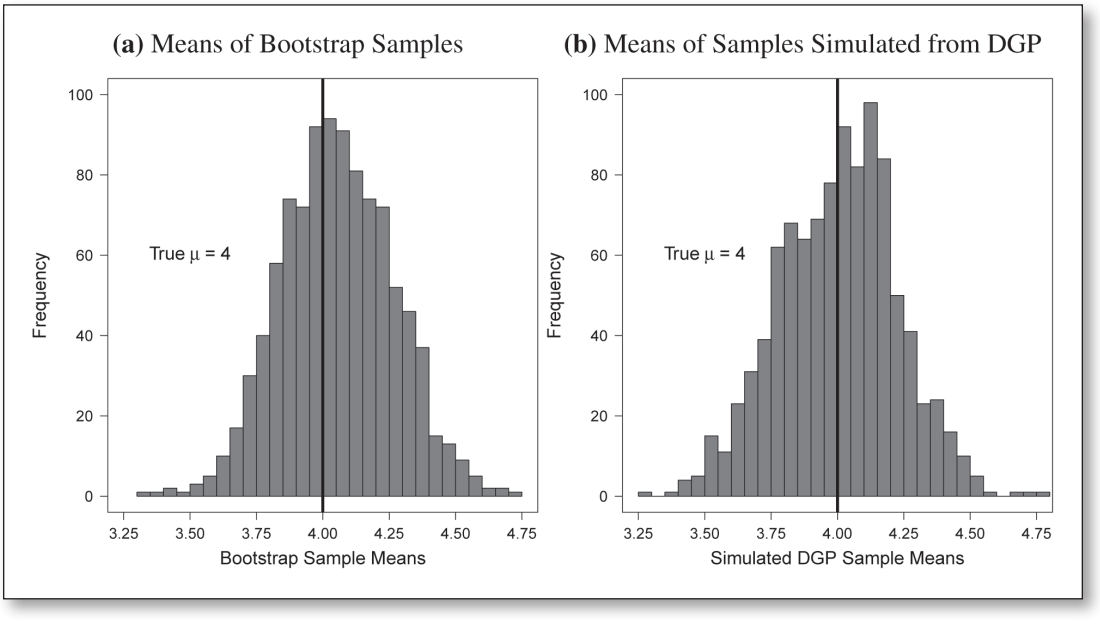
We show the results in Figure 8.3. Panel (a) plots a histogram of the 1,000 bootstrapped means, generated by resampling the observed data. Panel (b) plots a histogram of 1,000 means from repeatedly using `rnorm()` to draw from the true DGP. Notice that the two distributions look very similar; both are centered near the true mean of 4 and show about the same spread. In short, we can see that bootstrapping does mimic the process of drawing repeated samples from the statistic's sampling distribution, which in this case is the true DGP.[16]

To arrive at the bootstrap standard error of the mean, we can simply compute the standard deviation of the 1,000 bootstrap estimates of the mean. Notice that this produces an estimate that is very close to the standard error we computed using the formula above.

```
# Bootstrap estimate of the SE of the mean
boot.se <- sd(boot.b)
boot.se
[1] 0.2111522

se.b
[1] 0.2065029
```

Figure 8.3 Histograms of 1,000 Bootstrap Sample Means and Simulated Means



[16]This is due to the fact that the sample mean is an unbiased estimator of the population mean. If the estimator were biased, its sampling distribution would be different from the true DGP.

We can then compute a 95% confidence interval with the standard error as $\hat{\mu} \pm 1.96 \times \text{SE}_{\text{boot}}$.

```
# 95% confidence interval (parametric)
mean.b - 1.96*boot.se # Lower bound
[1] 3.648022

mean.b + 1.96*boot.se # Upper bound
[1] 4.475739
```

This is a parametric confidence interval because it uses properties of the normal distribution (i.e., the critical value of 1.96 multiplied by the standard error). For this to be appropriate, it must be reasonable to assume that the statistic of interest follows a normal distribution. For sample means, the central limit theorem makes this plausible, but this will not always be the case.

More generally, it strikes us as a bit odd to use a nonparametric resampling method to generate a simulated distribution of a parameter of interest, but then to use a parametric method to summarize the distribution of those parameter estimates. A completely distribution-free 95% confidence interval can be computed using the 2.5th and 97.5th quantiles of the bootstrap estimates.

```
# 95% confidence interval (nonparametric)
quantile(boot.b, .025) # Lower bound
    2.5%
3.656705

quantile(boot.b, .975) # Upper bound
  97.5%
4.47584
```

In this case, the two are very similar because the true DGP is a normal distribution. If the true DGP were not normal, these two methods would produce different results. Similarly, if the true DGP were not symmetric, the quantile method would capture that with a 95% confidence interval with bounds that were not equidistant from the mean. Thus, the quantile method (sometimes called the percentile method) is more flexible and better suited to recover a wider range of possible DGPs.

There are limits to the percentile method, however. First, the method requires you to have a fairly large sample of data so that you can be confident that the tails of the underlying population probability distribution are adequately represented in the sample data (Mooney & Duval, 1993). In other words, observations that are unlikely to appear given the true DGP may be completely unrepresented in a small sample of data rather than appearing rarely as they should. The percentile method can also be biased and inefficient (see Good, 2005). Still, if your original sample size is large, the percentile method is attractive because it does not impose any parametric assumptions.

We have discussed the two most common methods of generating confidence intervals via the bootstrap, but there are many others. Another problem with the percentile method is that you must assume that the distribution of the bootstrapped parameters of interest is an unbiased estimate of the true distribution of those parameters in the population. This is less restrictive than assuming the distribution must be normal, but still an assumption. In response, scholars have introduced a bias-corrected bootstrap confidence interval.[17]

Another approach to getting a proper confidence interval is to use a double-bootstrapping method to produce an estimate of the standard error of a statistic.[18] For this method, you still draw your large number of resamples to generate estimates of the parameter of interest. However, for each of those replicated samples, you perform another bootstrap on that replicated sample to generate an estimate of the standard error of that statistic. Rizzo (2008) offers a step-by-step discussion of how this procedure unfolds (section 7.4.4 starting on page 201). The main obstacle to this procedure is the computational time involved. If you draw 1,000 resamples to compute your parameters of interest, but also perform a bootstrap of 1,000 draws on each of those initial draws, you end up taking 1,000 × 1,000, or a total of 1,000,000 resampled draws from your data. If the statistical model you are estimating is even slightly computationally intensive, running it 1 million times will be very time-consuming.

The various methods described here for generating bootstrap confidence intervals (and a few others) are available to researchers in several `R` packages. These include the `boot` package, the `bootstrap` package, and the `bootcov()` function that is part of the `rms` package. We encourage interested readers to explore these packages and the publications we have cited if they are interested in learning more.

8.4.2 Bootstrapping With Multiple Regression Models

Bootstrapping is probably most useful to applied social scientists in the context of computing standard errors for multiple regression models. In the example below, we compute bootstrapped standard errors for the following model from the Ehrlich (1973) crime data used in Chapter 3.

```
set.seed(8873)
library(foreign)
crime <- read.dta("crime.dta")
```

[17]See Efron and Tibshirani (1993), Good (2005), Mooney and Duval (1993), or Rizzo (2008) for more discussion of this approach

[18]Good (2005) and Rizzo (2008) refer to this method as the bootstrap *t* interval, but Mooney and Duval (1993) call it the percentile *t* method.

```
# OLS model from the crime data
crime.1 <- lm(crime1960 ~ imprisonment + education + wealth +
+ inequality + population1960, data = crime)
summary(crime.1)

Call:
lm(formula = crime1960 ~ imprisonment + education + wealth +
    inequality + population1960, data = crime)

Residuals:
    Min      1Q  Median      3Q     Max
-525.23 -178.94  -25.09 145.62 771.65

Coefficients:
                  Estimate  Std. Error  t value   Pr(>|t|)
(Intercept)     -4213.3894   1241.3856   -3.394   0.001538 **
imprisonment    -3537.8468   2379.4467   -1.487   0.144709
education         113.6824     65.2770    1.742   0.089088 .
wealth              0.3938      0.1151    3.422   0.001420 **
inequality        101.9513     25.9314    3.932   0.000318 ***
population1960      1.0250      1.3726    0.747   0.459458
---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 297 on 41 degrees of freedom
Multiple R-squared: 0.4744,  Adjusted R-squared: 0.4103
F-statistic: 7.4 on 5 and 41 DF, p-value: 5.054e-05
```

There are several packages available to do bootstrapping in R, but we will start by writing our own because it helps develop intuition. The function we create takes two arguments: (1) the name of the model and (2) the number of bootstrap samples desired, which is set to a default of 1,000.[19] The first object created inside the function, `boot.est`, is an empty matrix to store the coefficient estimates from the bootstrap samples. It has a row for every bootstrap sample and a column for every coefficient estimate to be stored.[20] Next, the function starts a bootstrap loop that creates a new bootstrap sample each time.

```
# Inputs: the model name and the number of bootstrap samples
bootstrap <- function(model, n.boot){
# This creates an empty matrix for the bootstrap estimates
boot.est <- matrix(NA, nrow = n.boot, ncol = model$rank)
for(i in 1:n.boot){ # Start bootstrap loop
```

[19]Efron and Tibshirani (1993) contend that this number need not be large (e.g., 50–200). We recommend a large number such as 1,000 to improve precision (see also Rizzo, 2008).

[20]The object `rank` in a model object gives the number of coefficients estimated in that model.

Inside the bootstrap loop, the first step is to draw a sample of data from the observed data. As before, we use `sample()` to draw *n* observation numbers with replacement. Next, we create an empty matrix called `datai` that will be the data matrix of the bootstrap sample. We fill this matrix row by row in a second `for` loop. To do this, we use the `model` object that can be called from any estimation object in `R` (such as `lm()`). This object stores the entire data matrix used in estimation. We index the row numbers from `model` using the numbers sampled with replacement. At the end of this loop, `datai` is a matrix with *n* rows and a column for each variable in the model. Some observations from the original data may be repeated in multiple rows, and some observations may not appear at all. The final lines coerce this matrix into a data frame and assign the variables names.

```
# Select observations to go in bootstrap sample
boot.sample <- sample(1:nrow(model$model), replace = TRUE)
# Initiate bootstrap sample data
datai <- matrix(NA, nrow = length(boot.sample), ncol = model$rank)
for(j in 1:length(boot.sample)){ # Start data loop
datai[j, ] <- as.numeric(model$model[boot.sample[j], ]) # Coefficients from
                                                        # bootstrap sample
                                                        # j go into row
                                                        # j of datai
} # End bootstrap sample data
datai <- data.frame(datai)
colnames(datai) <- colnames(model$model)
```

The final task is to estimate the model on this bootstrapped sample and store the results. We use the `formula()` function to paste the model's formula and set the data to the object `datai`. We fill each row of `boot.est` with the coefficients. Once that is complete, the bootstrap loop is complete. The last step is to return the results. We include two objects in the `return()` function: (1) the actual bootstrap coefficient estimates in a matrix and (2) the variance–covariance matrix of those estimates, which we obtain with the `cov()` function.[21]

```
# Run the model on the bootstrap sample,
# then collect coefficients in boot.est matrix
boot.est[i, ] <- coef(lm(formula(model), data = datai))
cat("Completed", i, "of", n.boot, "bootstrap samples", "\\n")
} # End bootstrap loop

# Return the matrix of coefficient estimates and the
# bootstrapped variance-covariance matrix
return(list(boot.est = as.matrix(boot.est), boot.vcv = cov(boot.est)))
} # End function
```

[21]Note that this is a parametric means of estimating uncertainty. We could also compute confidence intervals for each coefficient with the quantiles of each set of bootstrap estimates.

We can then use the function on the crime data model and compare the results with the OLS standard errors. The `bootstrap()` function takes about 15 seconds to complete the process. In this case, the bootstrapped standard errors are the larger of the two for all of the coefficient estimates.

```
system.time( # Check how long it takes
crime.boot <- bootstrap(crime.1)
)
   user system elapsed
  14.71    0.03   14.78

crime.se <- cbind(coef(crime.1), sqrt(diag(vcov(crime.1))),
sqrt(diag(crime.boot$boot.vcv)))
colnames(crime.se) <- c("Coefficients", "Conventional SE", "Bootstrapped SE")
crime.se
                     Coefficients     Conventional SE      Bootstrapped SE
(Intercept)         -4213.3894135        1241.3856333         1549.2743323
imprisonment        -3537.8467727        2379.4467146         3093.9600328
education             113.6824207          65.2769966           81.4374504
wealth                  0.3937634           0.1150612            0.1516832
inequality            101.9513371          25.9314385           28.7289504
population1960          1.0250156           1.3725803            1.7785137
```

Increasing Computation Speed

Although our `bootstrap()` function works, it is somewhat slow because it relies on a `for` loop. A faster option is the `bootcov()` function in the `rms` package, which uses vectorized operations. To use the function, we first need to use the package's `ols()` function instead of `lm()` to estimate the model.[22]

```
library(rms)

crime.2 <- ols(crime1960 ~ imprisonment + education + wealth +
inequality + population1960, x = TRUE, y = TRUE, data = crime)
```

Then, we insert the model name into `bootcov()` and set the argument `B` to the number of bootstrap samples.

```
system.time( # Check how long it takes
crime.boot2 <- bootcov(crime.2, B = n.boot)
)
```

[22]The `rms` package also has several other GLM functions that work with `bootcov()`. The arguments `x = TRUE`, `y = TRUE` tell `R` to save the independent variables and dependent variable in a matrix, which the bootstrapping function requires.

```
  user system elapsed
  0.17   0.00    0.17

crime.boot2
Linear Regression Model

ols(formula = crime1960 ~ imprisonment + education + wealth +
    inequality + population1960, data = crime, x = TRUE, y = TRUE)

   n Model  L.R.       d.f.          R2     Sigma
  47       30.23          5      0.4744       297

Residuals:
    Min      1Q  Median      3Q     Max
-525.23 -178.94 -25.09 145.62 771.65

Coefficients:
                       Value    Std. Error         t     Pr(>|t|)
Intercept          -4213.3894     1565.4927   -2.6914    0.0102514
imprisonment       -3537.8468     3114.1249   -1.1361    0.2625267
education            113.6824       80.1067    1.4191    0.1634172
wealth                 0.3938        0.1518    2.5940    0.0130963
inequality           101.9513       28.2689    3.6065    0.0008343
population1960         1.0250        1.6787    0.6106    0.5448257

Residual standard error: 297 on 41 degrees of freedom
Adjusted R-Squared: 0.4103
```

Notice that `bootcov()` took less than 1 second to complete the operation—a substantial improvement over `bootstrap()`. Additionally, the standard errors produced by `bootcov()` look similar to those from `bootstrap()`, but they are not exactly the same. This highlights an important feature of bootstrapping. Because a different set of bootstrap samples could be drawn in each successive run of the code, the standard errors will be different each time. However, the analyst controls the number of repetitions, and adding more repetitions reduces the variation between calculations. Additionally, setting the seed beforehand will produce the same estimates each time.

Alternative Versions

We have focused primarily on bootstrapping by resampling complete observations, which is the most common approach. However, there are several alternatives that may also be useful. For example, one option is to resample residuals and assign them to observations. This comports better with the assumptions that the independent variables are fixed in repeated samples and that the error is what is random. Another possibility is an approach called the wild bootstrap, in which residuals are resampled and multiplied randomly by some number, such as 1 or – 1 or a random draw from a standard normal distribution. Wu (1986) shows that

this can improve bootstrap performance in the presence of heteroskedasticity. There are also several modifications to correct for bias if the distribution of bootstrap estimates is skewed (e.g., bias-corrected bootstrap or accelerated bootstrap, see Chernick & LaBudde, 2011; Efron, 1987; Good, 2005; Rizzo, 2008).

8.4.3 Adding Complexity: Clustered Bootstrapping

As we briefly mentioned above, the resampling methods we examine here can be adapted to several features of the data. For instance, the block bootstrap is designed to be used with time-series data (Künsch, 1989). Similarly, it is possible to resample groups of observations instead of individual observations if there is nonindependence due to clustering in the data. For example, recall the model on macroeconomic indicators in 14 countries from 1966 to 1990 from above. Each country appears in the data set multiple times. Because observations from each country likely share something in common that is not captured in the systematic component of the model, the residuals among observations clustered within each specific country are likely to be correlated with each other. We saw in Chapter 5 that this produces a downward bias in the standard error estimates we calculated. As an alternative, we can bootstrap the standard errors, resampling clusters of observations by country rather than individual observations. To do so, we make a small change to the code in the `bootcov()` function. First, here is the model with the conventional OLS standard errors (notice we removed the country indicator variables).

```
ols.macro2 <- ols(unem ~ gdp + capmob + trade
+ , x = TRUE, y = TRUE, data = macro)

ols.macro2
Linear Regression Model

ols(formula = unem ~ gdp + capmob + trade, data = macro, x = TRUE,
    y = TRUE)

   n Model   L.R.    d.f.         R2    Sigma
 350        118.8       3    0.2878    2.746

Residuals:
    Min      1Q  Median      3Q     Max
-5.3008 -2.0768 -0.3187 1.9789 7.7715

Coefficients:
            Value Std.      Error       t   Pr(>|t|)
Intercept     6.18129   0.450572  13.719  0.000e+00
gdp          -0.32360   0.062820  -5.151  4.355e-07
capmob        1.42194   0.166443   8.543  4.441e-16
trade         0.01985   0.005606   3.542  4.517e-04

Residual standard error: 2.746 on 346 degrees of freedom
Adjusted R-Squared: 0.2817
```

To bootstrap the standard errors by country, we use the `cluster` argument in the `bootcov()` function. This tells the function to resample countries instead of individual observations. In other words, if a country is drawn, all of its observations in the data enter the bootstrap sample rather than just one observation. If a country is resampled more than once, then all of its observations enter the sample more than once.

```
macro.boot <- bootcov(ols.macro2, B = n.boot, cluster = macro$country)

macro.boot
Linear Regression Model

ols(formula = unem ~ gdp + capmob + trade, data = macro, x = TRUE,
    y = TRUE)

    n Model  L.R.     d.f.         R2     Sigma
  350       118.8        3     0.2878     2.746

Residuals:
    Min      1Q  Median      3Q     Max
-5.3008 -2.0768 -0.3187 1.9789 7.7715

Coefficients:
                Value     Std. Error         t      Pr(>|t|)
Intercept      6.18129        1.35613     4.558     7.171e-06
gdp           -0.32360        0.09519    -3.399     7.541e-04
capmob         1.42194        0.54369     2.615     9.303e-03
trade          0.01985        0.01874     1.059     2.902e-01

Residual standard error: 2.746 on 346 degrees of freedom
Adjusted R-Squared: 0.2817
```

Notice that the bootstrapped standard errors (1.36, 0.10, 0.54, and 0.02) are considerably larger than the conventional OLS standard errors (0.45, 0.06, 0.17, and 0.01). This leads to the question of which standard error method is better. We can use simulation to produce an answer. Recall that in Chapter 5 we simulated clustered data and compared several different estimators for coefficient estimates and standard errors. We conduct a version of that simulation below and compare the OLS standard errors (OLS-SEs), robust cluster standard errors (RCSEs), and bootstrap cluster standard errors (BCSEs).[23]

The code for this simulation is a modification of the code from Chapter 5. We set the sample size to 200, the number of clusters to 25, and we set the independent variable, `X`, to vary at the cluster level. We estimate an OLS model, then compute the OLS-SE, RCSE, and BCSE.

[23]This is a short version of simulations done in Harden (2011).

```
# Simulation with BCSE
library(mvtnorm)
library(rms)
# Function to compute robust cluster standard errors (Arai 2011)
rcse <- function(model, cluster){
require(sandwich)
M <- length(unique(cluster))
N <- length(cluster)
K <- model$rank
dfc <- (M/(M - 1)) * ((N - 1)/(N - K))
uj <- apply(estfun(model), 2, function(x) tapply(x, cluster, sum))
rcse.cov <- dfc * sandwich(model, meat = crossprod(uj)/N)
return(rcse.cov)
}

set.seed(934656) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.cluster <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store
                                                     # the estimates
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 200 # Sample size
p <- 0.5 # Rho
nc <- 25 # Number of clusters
c.label <- rep(1:nc, each = n/nc) # Cluster label

for(i in 1:reps){ # Start the loop
i.sigma <- matrix(c(1, 0, 0, 1 - p), ncol = 2) # Level 1 effects
i.values <- rmvnorm(n = n, sigma = i.sigma)
effect1 <- i.values[ , 1]
effect2 <- i.values[ , 2]

c.sigma <- matrix(c(1, 0, 0, p), ncol = 2) # Level 2 effects
c.values <- rmvnorm(n = nc, sigma = c.sigma)
effect3 <- rep(c.values[ , 1], each = n/nc)
effect4 <- rep(c.values[ , 2], each = n/nc)

X <- effect3 # X values unique to level 2 observations
error <- effect2 + effect4

Y <- b0 + b1*X + error # True model

model.ols <- lm(Y ~ X) # Model estimation

vcv.ols <- vcov(model.ols) # Variance-covariance matrices
vcv.rcse <- rcse(model.ols, c.label)
vcv.bcse <- bootcov(ols(Y ~ X, x = TRUE, y = TRUE),
B = n.boot, cluster = c.label)
```

```
par.est.cluster[i, 1] <- model.ols$coef[2] # Coefficients
par.est.cluster[i, 2] <- sqrt(diag(vcv.ols)[2])
par.est.cluster[i, 3] <- sqrt(diag(vcv.rcse)[2])
par.est.cluster[i, 4] <- sqrt(diag(vcv.bcse$var)[2])
cat("Just completed iteration", i, "\n")
} # End the loop
```

Next, we compute the coverage probabilities of each standard error method.

```
ols.cp <- coverage(par.est.cluster[ , 1], par.est.cluster[ , 2], b1,
 df = n - model.ols$rank)
ols.cp$coverage.probability
[1] 0.641

rcse.cp <- coverage(par.est.cluster[ , 1], par.est.cluster[ , 3], b1,
 df = n - model.ols$rank)
rcse.cp$coverage.probability
[1] 0.909

bcse.cp <- coverage(par.est.cluster[ , 1], par.est.cluster[ , 4], b1,
 df = n - model.ols$rank)
bcse.cp$coverage.probability
[1] 0.923
```

As we saw in Chapter 5, the OLS-SE is severely biased downward, with a coverage probability of 0.641. Additionally, while RCSE is better, it still shows evidence of being too small, with a coverage probability of 0.909 and simulation error bounds of [0.891, 0.927]. Finally, bootstrapping by cluster performs the best of the three, though it is still biased slightly downward. The BCSE coverage probability is 0.923 with error bounds [0.906, 0.940]. Overall, among these alternatives, bootstrapping by cluster provides the best method of estimating standard errors in the presence of clustered data (for more details, see Harden, 2011).[24] This example shows both the benefits of bootstrapping as a method of estimating standard errors and the use of a simulation to evaluate competing methods.

8.5 CONCLUSIONS

Resampling methods are similar to simulation in that they use an iterative process to summarize the data. The main difference is that they rely on the observed sample of data rather than a theoretical DGP. This empirical basis typically gives resampling methods robustness to assumption violations, as shown by the jackknife estimate of standard errors in the presence of heteroskedasticity, the BCSE

[24]The BCSE method can also be extended to multilevel model standard errors (Harden, 2012a).

performance with clustered data, and the quantile method of computing bootstrap confidence intervals. Data from the real world are rarely perfectly well behaved, so we recommend that researchers consider using resampling methods in their own work.

Within the various methods, there are cases where each one may be most appropriate. Permutation and randomization testing are typically best for experimental data where there is a clear treatment variable and null hypothesis of no effect. Randomization testing has been extended to the multiple regression framework (e.g., Erikson et al., 2010), but we do not recommend it because it does not allow for estimation of the full variance–covariance matrix of the coefficient estimates. Jackknifing is a good option when the analyst is concerned about the undue influence of particular data points. However, it performs poorly in small samples and when the statistic of interest is not smooth.

We recommend bootstrapping for most cases because it is flexible and robust to many different types of data. It also allows for estimation of the full covariance matrix in a multiple regression model while also generally performing well in small samples (Efron & Tibshirani, 1993). Moreover, its strong connection to the conceptualization of simulating repeated samples makes the method intuitively appealing. In short, while not a panacea, bootstrapping is a very useful tool for applied social scientists.

Having completed chapters on simulation and on resampling methods, we bring the two together in the next chapter. First, we look at a method for simulating from the observed data as a means of generating quantities of interest from model results. Then, we discuss cross-validation as a "resampling-like" tool for evaluating the fit of statistical models.