Designing Monte Carlo Simulations in R

James E. Pustejovsky and Luke W. Miratrix

2022-05-10

Contents

Chapter 1

Introduction

In this text we present an approach for writing Monte Carlo simulations in R. Monte Carlo simulations are an essential tool of inquiry for quantitative methodologists and students of statistics, useful both for small-scale or informal investigations and for formal methodological research. Our focus in this text is on the best practices of simulation design and how to use simulation to be a more informed and effective quantitative analyst.

The primary purpose of this monograph is to provide a guide to designing simulation studies to answer questions about statistical methodology. The focus is on studies used for formal research purposes (i.e., as might appear in a journal article or dissertation), although many of the programming techniques are relevant to less formal situations as well. This guide also covers other uses of simulation, such as for power analysis or even statistical inference.

In general, simulation studies allow for investigating the performance of a statistical model or method under known data-generating processes. By controlling the data generating process (e.g., by specifying true values for the parameters of a statistical model) and then repeatedly applying a statistical method to data generated by that process, it becomes possible to assess how well a statistical method works.

Overall, we will show how simulation frameworks allow for rapid exploration of the impact of different design choices and data concerns, and how simulation can answer questions that are hard to answer using direct computation (e.g., with power calculators or mathematical formula). Simulations are particularly useful for studying models and estimation methods where relevant algebraic formulas are not available, not easily applied, or not sufficiently accurate. For example, available algebraic formulas are often based on asymptotic approximations, which might not "kick in" if sample sizes are moderate. This is, for example, a particular concern with hierarchical data structures that include only 20 to 40 clusters, which is the range of common sample sizes in many large-scale

randomized trials in education research.

1.1 Some of simulation's many uses

As alluded to above, simulation is can be useful across a wide range of areas. Some are the focus of this text, and others can be more easily tackled with the tools we present. But to wet the appetite, consider the following areas where one might find need of simulation.

1.1.1 Comparing statistical approaches

Comparing statistical approaches is perhaps the most common use of Monte Carlo simulation. In the statistical methodology literature, for example, authors will frequently use simulation to compare their newly proposed method to more traditional approaches to make a case for their method being of real value. Other simulation-based research will often work to align a literature by systematically comparing a suite of methods all designed to achieve a given task to one another. In the best case, simulation can show how trade-offs between methods can occur in practice.

For a classic example, Brown and Forsythe (1974) compared four different procedures for conducting a hypothesis test for equality of means in several populations (i.e., one-way ANOVA), but where the population variances are not equal. Overall, simulation can be critical for understanding the benefits and drawbacks of analytic methods in practice.

Comparitive simulation can also have a practical application: In many situations, more than one modeling approach is possible for addressing the same research question (or estimating the same target parameter). Comparing the costs of one vs. another using simulation is informative for guiding the design of analytic plans (such as plans included in pre-registered study protocols). As an example of the type of questions that researchers might encounter in designing analytic plans: what are the benefits and costs of using a model that allows for cross-site impact variation for a multi-site trial in practice?

1.1.2 Assessing performance of complex pipelines

In practice, statistical methods are often used in combination. For instance, in a regression model, one could first use a statistical test for heteroskedasticity (e.g., the White test or the Breusch-Pagan test) and then determine whether to use conventional standard errors or HRSE depending on the result of the test. This combination of an initial diagnostic test followed by contingent use of different statistical procedures is all but impossible to analyze mathematically,

but it is straight-forward to simulate (see, for example, Long & Ervin, 2000). In particular, with simulation, we can verify a proposed pipeline is *valid*, meaning that the conclusions it draws are correct at a given level of certainty.

Simulating an analytic pipeline can be used for statistical inference as well. With bootstrap or parametric bootstrap approaches, for example, one is, in essence, repeatidly simulating data and putting it through an entire analytic pipeline to assess how stable estimation is. How much a final point estimate varies across the simulation trials is the standard error for the context being simulated; an argument by analogy is what connects this to inference on the original data and point estimate.

1.1.3 Assessing performance under misspecification

Many statistical estimation procedures can be shown (through mathematical analysis) to perform well when the assumptions they entail are correct. However, it practice it is often of interest to also understand their robustness—that is, their performance when one or more of the assumptions is incorrect. For example, how important is the normality assumption underlying multilevel modeling? What about homoskedasticity?

In a similar vein, when the true data-generating process meets stringent assumptions (e.g., constant treatment effect), what are the potential gain of exploiting such structure in the estimation process? Conversely, what are the costs of using flexible methods that do not impose the stringent assumption? A researcher designing an analytic plan would want to be well informed of such tradeoffs in the context they are working in. Simulation allows for such investigation and comparison.

1.1.4 Assessing the finite sample performance of a statistical approach

Many statistical estimation procedures can be shown (through mathematical analysis) to work well *asymptotically*—that is, given an infinite amount of data—but their performance in small samples is more difficult to quantify. Simulation is a tractable approach for assessing the small-sample performance of such methods, or for determining minimum required sample sizes for adequate performance of a method. This is perhaps one of the most important uses for simulation: mathematical theory generally is asymptotic in nature, but we are living in the finite world and practice. In order to know whether the asymptotics "kick in" we must rely on simulation.

For example, heteroskedasticity-robust standard errors (HRSE) are known to work asymptotically, but can be misleading in small samples. Long and Ervin (2000) use extensive simulations to investigate the properties of different heteroscedasticity robust standard error estimators for linear regression across a

range of sample sizes, demonstrating that the most commonly used form of these estimators often does NOT work well with sample sizes typical in the social sciences. Simulation could answer what asymptotics could not: how these estimators work in typical practice.

For another example, recent work has developed the Fixed-Intercept, Random Coefficient method for estimating and accounting for cross site treatment variation in multisite trials. When there are a moderate number of clusters it appears that the numerical (asymptotic based) estimates of performance are not very accurate. Simulation can unpack these trends and give a more accurate picture of effectiveness in these real contexts.

1.1.5 Conducting Power Analyses

By repeatidly simulating and then analyzing data from a guessed-at world, a researcher can easily calculate the power to detect the effects so modeled, if that world were true. This can allow for power analyses far more nuanced and tailored to a given circumstance than typical power calculators. In particular, simulation can be useful for the following:

- Available formulas for power analysis in multi-site block- or cluster-randomized trials (such those implemented in the Optimal Design and PowerUp! Software) assume that sites are of equal size and that outcome distributions are unrelated to site size. Small deviations from these assumptions are unlikely to change the results, but in practice, researchers may face situations where sites vary quite widely in size or where site-level outcomes are related to site size. Simulation can estimate power in this case.

- Available software (such as PowerUp!) allows investigators to make assumptions about anticipated rates of attrition in cluster-randomized trials, under the assumption that attrition is completely at random. However, researchers might anticipate that attrition will be related to baseline characteristics, leading to data that is missing at random but not completely at random. How will this affect the power of a planned study?

- There are some closed-form expressions for power to test mediational relations (i.e., indirect and direct effects) in a variety of different experimental designs, and these formulas are now available in PowerUp!. However, the formulas involve a large number of parameters (including some where it may be difficult in practice to develop credible assumptions) and they apply only to a specific analytic model for the mediating relationships. Researchers planning a study to investigate mediation might therefore find it easier to generate realistic data structures and conduct power analysis via simulation.

1.1.6 Simulating processess

Less central to this book, but a very common use for simulation, is to simulation some sort of complex process to better understand it or the consequences of it. For example, some larger school districts (e.g., New York City) have centralized lotteries where families rank some number of schols by order of preference. The central office then assigns students to schools via a lottery procedure where each student gets a lottery number that breaks ties when there are too many students desiring to go to a specific school. As a consequence, students have a random probability of assignment to the schools on their list, depending on their choices, the choices of other students, and their lottery numbers.

We can use this process to estimate the causal impact of being assigned to one school vs. another, but only if we have those probabilities of school assignment. We can obtain them via simulation: we repeatidly run the school lottery over and over, and record where everyone gets assigned. Using these final propensity scores we can move forward with our analysis.

Another famous area of process simulation are climate models, where researchers simulate the process of climate change. These physical simulations mimic very complex systems to try and understand how perturbations (e.g., more carbon release) will impact downstream trends.

1.2 The perils of simulation as evidence

As the above argues, simulation has the potential to be a powerful tool for investigating quantitative methods. Unfortunately, simulation-based argument also opens up a large can of worms, and is very susceptible to critique. These critiques usually revolve around what the data generating process of the simulations is. Are the simulated data realistic? Was the simulation systematic in exploring a wide variety of scenarios, allowing for truly general conclusions?

The best way to answer these arguments is through transparency: explicitly state what was done, and provide code so people can tweak it to run their own simulations. Another important component of a robust argument is systematic variation: design one simulations so that one can easily simulate across a range of scenarios. Once that is in place, systematically explore myriad scenarios and report all of the results.

Due to the flexibility in the design of simulations, they are held in great skepticism by many. A summary of this is the motto

> Simulations are doomed to succeed.

Simulations are alluring: once a simulation framework is set up, it is easy to tweak and adjust. It is natural for us all to continue to do this until the simulation works “as it should.” This means, if our goal was to show something we

"know" is right (e.g., that our new estimation procedure is better than another), we will eventually find a way to align our simulation with our intuition. This is, simply put, a version of fishing.

To counteract that, challenge yourself to design scenarios where things do not work as you expect. Try to learn the edges that separate where things work, and where things do not.

1.3 Why R and RStudio?

The statistical software package R runs on both PCs and Macs. The software is free and available online. R is straightforward to learn, but is sufficiently powerful and versatile to be useful for real projects that you might carry out. It is used widely for statistical work, and frequently used in research in such fields as education, psychology, economics, medical research, epidemiology, public health, and political science.

We highly recommend using RStudio, which makes using R easier. RStudio is an Integrated Development Environment (IDE) that structures your experience, helps keep things organized, and offers multiple time-saving features to make your programming experience better. You might also consider using R Markdown. R Markdown allows for generating documents with embedded R code and R output in a clean format, which can greatly help report generation.

Many people seem to believe that R is particularly technically challenging and difficult to master. This probably stems from its extreme flexibility; it is a fully functional programming language as well as a statistical analysis package. R can do things that many other software packages (we're looking at you, Stata) essentially can't. But these more involved things are frequently hard to do because they require you to think like a statistical programmer rather than a data analyst. As a result, R is perceived as a "hard" language to use. For simulation, in particular, the ability to easily write functions (bundles of commands that you can easily call in different manners), have multiple tables of data in play at the same time, and leverage the vast array of other people's work all make R an attractive option.

Fundamentally, however, we believe even for straightforward, off-the-shelf analyses, R is arguably easier to learn and use in many ways than programs such as Stata or SAS. This is especially true when R is combined with RStudio for a better programming experience.

1.3.1 Functions

A critical component of simulation design is the use of functions. A function is a bundle of commands that you can name, so you can use those commands

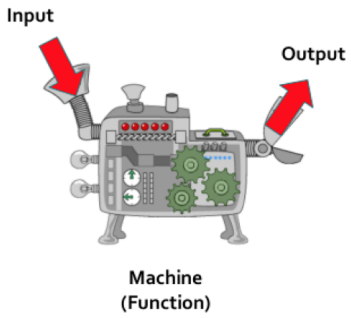


Figure 1.1: A function as a machine

over and over. You can think of it as a machine, with a hopper that takes some inputs, and a chute that spits out an output based on those inputs. A function can do anything, and it can even be random in its behavior. For example, the `rnorm()` function in R takes a number, and gives you that many random, normally distributed, numbers in response. See Chapter 19 of R for Data Science for an extended discussion, but here is an example function to get you started:

```
one_run <- function( N, mn ) {
  vals = rnorm( N, mean = mn )
  tt = t.test( vals )
  tt$p.value
}
```

The above makes a new command, `one_run()` that takes a sample size N and mean `mn` and generates N normally distributed points centered on `mn`, conducts a t-test on the results, and returns the p-value for testing whether the mean is zero or not.

We then call our new method as so:

```
one_run( 100, 0.2 )
```

```
## [1] 0.108358
```

```
one_run( 10, 0.3 )
```

```
## [1] 0.1881897
```

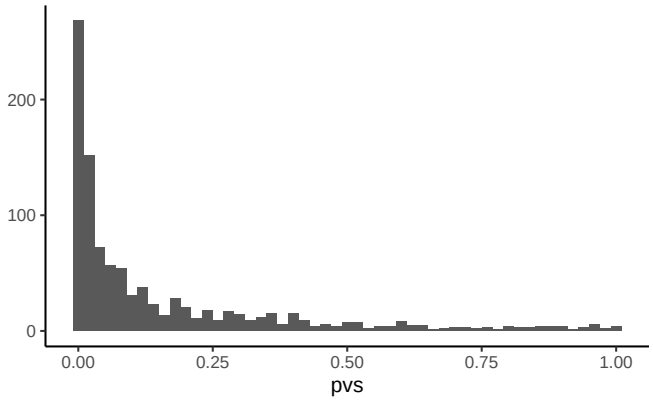
```
one_run( 10, 0.3 )
```

```
## [1] 0.0150982
```

In this case, each time we run our code, we get a different answer since we are generating random numbers with each call.

We can call it a lot, like so:

```
set.seed( 30303 )
pvs = replicate( 1000, one_run( 100, 0.2 ) )
qplot( pvs, binwidth=0.02 )
```



We see that if our sample size is 100, and the true mean is 0.2, we often get low p-values, but not always. We can calculate the power of our test as so:

```
mean( pvs <= 0.05 )
```

```
## [1] 0.492
```

Via simulation, we have discovered we have about a 49% chance of rejecting the null, if the alternative is 0.2 and our sample size is 100.

Basically the rest of the book is an elaboration of the ideas above.

1.3.2 A dangerous function

Functions are awesome, but if you violate their intention, you can get into trouble. For example, consider the following script you might have:

```
secret_ingredient <- 3

# blah blah blah

funky <- function(input1, input2, input3) {

  # do funky stuff
  ratio <- input1 / (input2 + 4)
  funky_output <- input3 * ratio + secret_ingredient

  return(funky_output)
}

funky(3, 2, 5)
```

```
## [1] 5.5
```

You then call it like so:

```
secret_ingredient <- 100
funky(3, 2, 5)
```

```
## [1] 102.5
```

This is bad: our function acts differently even when we give it the same arguments. Such behavior can be quite confusing, as we generally expect the function to work a certain way, given the inputs we provided it.

Even worse, we can get errors depending on this extra feature:

```
secret_ingredient <- "A"
funky(3, 2, 5)
```

```
## Error in input3 * ratio + secret_ingredient: non-numeric argument to binary operator
```

This is the #1 gotcha with function writing. Be careful to, in a function, only use what you are *passed*, as in only use those parameters that are specified at the head of the function. It is easy to write terrible, confusing code in R.

You can fix it by *isolating the inputs*:

```
secret_ingredient <- 3

# blah blah blah

funky <- function(input1, input2, input3, secret_ingredient) {

  # do funky stuff
  ratio <- input1 / (input2 + 4)
  funky_output <- input3 * ratio + secret_ingredient

  return(funky_output)
}

funky(3, 2, 5, 3)
```

```
## [1] 5.5
```

Now things are nice:

```
secret_ingredient <- 100
funky(3, 2, 5, 3)
```

```
## [1] 5.5
```

```
funky(3, 2, 5, 100)
```

```
## [1] 102.5
```

1.3.3 Function skeletons

When we say "skeleton" we simply mean the header of a function, without the middle stuff. E.g.,

```
run_simulation <- function( N, J, mu, sigma, tau ) {

}
```

These are useful for sketching out a general plan of how to organize code.

1.3.4 %>% (Pipe) dreams

We extensively use the "pipe" in our code. For those unfamiliar, we here spend a moment discussing it, but see R for Data Science, Chapter 18, for more. The %>% command allows you to apply a **sequence of functions** to a data frame; this makes your code read like a story book.

With conventional code we have

```
res1 <- f(my_data, a = 4)
res2 <- g(res1, b = FALSE)
result <- h(res2, c = "hot sauce")
```

Or

```
result <- h(g(f(my_data, a = 4),
              b = FALSE),
            c = "hot sauce")
```

Ouch.

With the pipe we have

```
result <-
  my_data %>%         # initial dataset
  f(a = 4) %>%        # do f() to it
  g(b = FALSE) %>%    # then do g()
  h(c = "hot sauce") # then do h()
```

Nice!

Chapter 2

An initial simulation

We begin with the concrete before we get abstract, with an initial simulation study that looks at the t-test under violations of the normality assumption. The goal here is to make the idea of Monte Carlo simulation concrete, and to illustrate the idea of replication and aggregation of results. In a sense, this chapter is the entire book. That being said, we hope to provide some deeper thinking on all the component parts in everything that follows.

This simulation (and much of this book) relies on the `tidyverse` package, which needs to be loaded. We also shut up tidyverse's weird summarize warnings while we are at it. These lines of code are pretty much the header of any script we use.

```
library( tidyverse )
options(list(dplyr.summarise.inform = FALSE))
```

We use methods from the "tidyverse" for cleaner code and some nice shortcuts. See the online, free and excellent (https://r4ds.had.co.nz/)[R for Data Science textbook] for more on the tidyverse.

We will start with a simulation study to examine the coverage of the lowly one-sample t-test. *Coverage* is the chance of a confidence interval capturing the true parameter value.

Let's first look at the t-test on some fake data:

```
# make fake data
dat = rnorm( 10, mean=3, sd=1 )

# conduct the test
tt = t.test( dat )
tt
```

```
##
##  One Sample t-test
##
## data:  dat
## t = 9.6235, df = 9, p-value = 4.92e-06
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  2.082012 3.361624
## sample estimates:
## mean of x
##  2.721818
```

```
# examine the results
tt$conf.int
```

```
## [1] 2.082012 3.361624
## attr(,"conf.level")
## [1] 0.95
```

For us, we have a true mean of 3. Did we capture it? To find out, we use `findInterval()`

```
findInterval( 3, tt$conf.int )
```

```
## [1] 1
```

`findInterval()` checks to see where the first number lies relative to the range given in the second argument. E.g.,

```
findInterval( 1, c(20, 30) )
```

```
## [1] 0
```

```
findInterval( 25, c(20, 30) )
```

```
## [1] 1
```

```
findInterval( 40, c(20, 30) )
```

```
## [1] 2
```

So, for us, `findInterval == 1` means we got it! Packaging the above gives us the following code:

```
# make fake data
dat = rnorm( 10, mean=3, sd=1 )

# conduct the test
tt = t.test( dat )

# evaluate the results
findInterval( 3, tt$conf.int ) == 1
```

```
## [1] TRUE
```

2.1 Simulation for a single scenario

The above shows the canonical form of a single simulation trial: make the data, analyze the data, decide how well we did. Before writing a simulation, it is wise to understand the thing we plan on simulating. Mucking around with code like this gets us ready.

Now let's look at coverage by doing the above many, many times and seeing how often we capture the true parameter:

```
rps = replicate( 1000, {
  dat = rnorm( 10 )
  tt = t.test( dat )
  findInterval( 0, tt$conf.int )
})
table( rps )
```

```
## rps
##   0   1   2
##  27 957  16
```

```
mean( rps == 1 )
```

```
## [1] 0.957
```

The `replicate()` function is one of many ways we can do things over and over in R. We got about 95% coverage, which is good news. We can also assess *simulation uncertainty* by recognizing that our simulation results are an i.i.d. sample of the infinite possible simulation runs. We analyze this sample to see a range for our true coverage.

```
hits = as.numeric( rps == 1 )
prop.test( sum(hits), length(hits), p = 0.95 )
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  sum(hits) out of length(hits), null probability 0.95
## X-squared = 0.88947, df = 1, p-value = 0.3456
## alternative hypothesis: true p is not equal to 0.95
## 95 percent confidence interval:
##  0.9420144 0.9683505
## sample estimates:
##     p
## 0.957
```

We have no evidence that our coverage is not what it should be: 95%.

Things working out should hardly be surprising. The t-test is designed for normal data and we generated normal data. In other words, our test is following theory when we meet our assumptions. Now let's look at an exponential distribution to see what happens when we don't have normally distributed data. We are simulating to see what happens when we voilate our assumptions behind the t-test. Here, the true mean is 1 (the mean of a standard exponential is 1).

```
rps = replicate( 1000, {
  dat = rexp( 10 )
  tt = t.test( dat )
  findInterval( 1, tt$conf.int )
})
table( rps )
```

```
## rps
##   0   1   2
##   3 905  92
```

Our interval is often entirely too low and very rarely does our interval miss because it is entirely too high. Furthermore, our average coverage is not 95% as it should be:

```
mean( rps == 1 )
```

```
## [1] 0.905
```

Again, to take simulation uncertainty into account we do a proportion test. Here we have a confidence interval of our true coverage under our model misspecification:

```
hits = as.numeric( rps == 1 )
prop.test( sum(hits), length(hits) )
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  sum(hits) out of length(hits), null probability 0.5
## X-squared = 654.48, df = 1, p-value < 2.2e-16
## alternative hypothesis: true p is not equal to 0.5
## 95 percent confidence interval:
##  0.8847051 0.9221104
## sample estimates:
##     p
## 0.905
```

Our coverage is *too low*. Our t-test based confidence interval is missing the true value (1) more than it should.

2.2 Simulating across different scenarios

The above gives us an answer for a single, specific circumstance. We next want to examine how the coverage changes as the sample size varies. So let's do a one-factor experiment, with the factor being sample size. I.e., we will conduct the above simulation for a variety of sample sizes and see how coverage changes.

We first make a function, wrapping up our *specific, single-scenario* simulation into a bundle so we can call it under a variety of different scenarios.

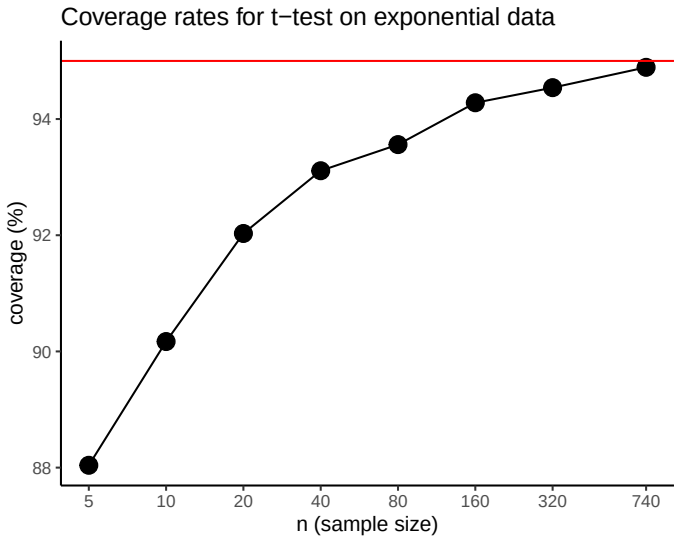
```
run.experiment = function( n ) {
  rps = replicate( 10000, {
    dat = rexp( n )
    tt = t.test( dat )
    findInterval( 1, tt$conf.int )
  })

  mean( rps == 1 )
}
```

Now we run `run.experiment` for different n. We do this with `map_dbl()`, which takes a list and calls a function for each value in the list (See R for DS, Chapter 21.5).

```
ns = c( 5, 10, 20, 40, 80, 160, 320, 740 )
cover = map_dbl( ns, run.experiment )
```

We next take our results, make a data.frame out of them, and plot:

```
res = data.frame( n = ns, coverage=cover )
ggplot( res, aes( x=n, y=100*coverage ) ) +
  geom_line() + geom_point( size=4 ) +
  geom_hline( yintercept=95, col="red" ) +
  scale_x_log10( breaks=ns ) +
  labs( title="Coverage rates for t-test on exponential data",
        x = "n (sample size)", y = "coverage (%)" )
```



Note the plot is on a log scale for the x-axis.

So far we have done a very simple simulation to assess how well a statistical method works in a given circumstance. We have run a single factor experiment, systematically varying the sample size to examine how the behavior of our estimator changes. In this case, we find that coverage is poor for small sample sizes, and still a bit low for higher sample sizes is well.

More broadly, the overall simulation framework, for a given scenario, is to repeatedly do the following:

- Generate data according to some decided upon data generation process (DGP). This is our model.

- Analyze data according to some other process (and possibly some other assumed model).
- Assess whether the analysis "worked" by some measure of working (such as coverage).

Frequently we would analyze our data with different methods, and compare performances across the methods. We might do this, for example, if we were trying to see how our new, nifty method we just invented compares to business as usual. We would also want to vary multiple aspects of our simulation (which we call factors), such as exploring coverage across a range of sample sizes (as we did) and also different disributions for the data (e.g., normal, exponential, t, and so forth). In the next chapter we provide a framework for simulation studies, building on the core arc of this example.

Chapter 3

Structure of a simulation study

In the prior chapter we saw a simple simulation evaluation of a t-test. We next break that simulation down into components, and then in the subsequent chapters we dig into how to think about each component.

3.1 General structure of a simulation

The easiest way to evaluate an estimator is to generate some data from scratch, generating it in such a way that we know what the "right answer'' is, and then to analyze our data using our estimators we wish to study. We then check to see if the estimators got the right answer. We can write down how close the estimators got, whether the estimators were too high or too low, and so forth. We can also use the estimators to conduct a hypothesis test, testing the hypothetical null, and we can write down whether we rejected or did not reject this null.

If we do this once, we have some idea of whether the estimators worked in that specific example, but we don't know if this is due to random chance. To assess general trends, therefore, we repeat this process over and over, keeping track of how well our estimators did in each trial. We finally aggregate our results, and see if one estimator systematically outperformed the other.

If we want to know if an estimator is generally superior to another, we can generate data from a variety of different scenarios, seeing if the estimator is systematically winning across simulation contexts. If we do this in a a structured and thoughtful manner, we can eventually make claims as to the behaviors of our estimators that we are investigating.

A simulation study can be thought of as something like a controlled scientific experiment: we want to understand the properties of our estimators, so we put them in a variety of different scenarios to see how they perform. We then look for general trends across these scenarios in order to understand general patterns of behavior of our estimators.

In particular, a simulation would be the following steps:

1. For a (the Data Generation Process), repeat R times:
   (a) Generate a dataset with a known target parameter.
   (b) Estimate this parameter with each of our estimators.
   (c) Record these estimates, and how close these estimates are to the target.
2. Aggregate our R trials for each estimator to assess how well our estimators work in practice.

We might then systematically repeat the above for a series of different scenarios to generalize our findings.

As we saw in our initial example, the logic of simulation is, for a specific and specified scenario scenario:

1. **Generate** a sample of data based on a specified statistical model/process.
2. **Analyze** data using one or more procedures/workflows.
3. **Repeat** (1) & (2) R times.
4. **Summarize** the performance of the procedure across our R repetitions.

We will then do this for a series of scenarios, so we can see how performance changes as we chance circumstance. But first, let's just focus on a single scenario.

3.2 Tidy simulations

In general, we advocate for writing *tidy simulations*, meaning we keep all the components of a simulation separate. The main way to keep things tidy is to follow a **modular approach**, in which each component of the simulation is implemented *as a separate function* (or potentially a set of several functions). Writing separate functions for the different components of the simulation will make the code easier to read, test, and debug. Furthermore, it makes it possible to swap components of the simulation in or out, such as by adding additional

estimation methods or trying out a data-generating model that involves different distributional assumptions. In particular, we write separate functions for each step of the simulation, and then wire these functions together at the end. In code, we can start with the skeletons of:

```
# Generate
generate_data <- function(params) {
  # stuff
}

# Analyze
analyze <- function(data) {
  # stuff
}

# Repeat
one_run <- function() {
  dat <- generate_data(params)
  analyze(dat)
}
results <- rerun(R, one_run())

# Summarize
assess_performance <- function(results) {
  # stuff
}
```

To repeat, this approach has several advantages:

- Easier to check & debug.
- Easier to modify your code.
- Easier to make everything run fast.
- Facilitates creative re-use.

In fact, the `simhelpers` package will build these skeletons (along with some other useful code to wire the pieces together) via the `create_skeleton()` method.

```
simhelpers::create_skeleton()
```

Starting with this, you will already be well on the road to writing a tidy simulation.

3.2.1 Data-generating model

The data-generating model takes a set of parameter values as input and generates a set of simulated data as output. When we generate data, we control the ground truth! This allows us to know what the answer is, so we know if our methods are doing the right thing.

3.2.2 Estimation methods

The estimation methods consist of the set of statistical procedures under examination. Each method takes a dataset and produces a set of estimates or results (i.e., point estimates, standard errors, confidence intervals, p-values, etc.). You might have different functions for each estimation method you are investigating. Our estimation methods should, in principle, work on real data as well as simulated data; the more we can "black box" these into a single function call, the easier it will be to separate out the structure of the simulation and the complexity of the methods being evaluated.

3.2.3 Repetition

There are a variety of ways in R to do something over and over. The one above, `rerun` does exactly what it sounds like: repeatidly run the line of code a given number of times.

Making a helper method such as `one_run()` makes debugging our simulations a lot, lot easier. This `one_run()` method is like the coordinator or dispatcher of our system: it generates the data, calls all the evaluation methods we want to call, combines all the results, and hands them back for recording.

We will usually stack all our returned results into one large dataframe of simulation results to ready it for the next step, which is assessing performance.

3.2.4 Performance summaries

Performance summaries are the metrics used to assess the performance of a statistical method. Interest usually centers on understanding the performance of a method over repeated samples from a data-generating process. For example, we might want to know how close our estimator gets to the target parameter, on average. Or we might want to know if a confidence interval captures the truth the right amount of time. To estimate these quantities we repeat steps 2 and 3 many times to get a large number of simulated estimates. We then *summarize the distribution* of the estimates to characterize performance of a method.

3.2.5 Results

Simulation results consist of sets of performance metrics for every combination of parameter values in the experimental design. Thus, steps 2 through 4 will need to get repeated a bunch.

3.3 Multiple Scenarios

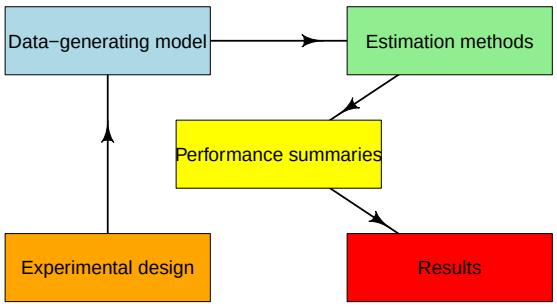
The above gives a breakdown for running a simulation for a single context. In our t-test case study, for example, we might ask how well the t-test works when we have $n = 100$ units and an exponential distribution to our data. But we rarely want to example a single context, but instead want to explore how well a procedure works across a range of contexts.

We again use the principles of modular coding: we write code to do a single scenario (and wrap that in a function), and then call that function for all the scenarios we wish. This is a type of *designed experiment*, in which factors such as sample size and true parameter values are systematically varied. In fact, simulation studies typically follow a **full factorial design**, in which each level of a factor (something we vary, such as sample size, true treatment effect, or residual variance) is crossed with every other level. The experimental design then consists of sets of parameter values (including design parameters, such as sample sizes) that will be considered in the study. We discuss this after we more fully develop the core concepts listed above.

3.4 Keeping the pieces organized

The above presents a simulation study as having five main components:

```
## Warning: `data_frame()` was deprecated in tibble 1.1.0.
## Please use `tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```



The code we write to implement simulations should follow this same structure, with different functions corresponding to each component.

In our methodological work, we try to always follow the same workflow when writing simulations. We make no claim that this is the only or best way to do things, but it does work for us.

To write a simulation we start by making a template via the `simhelpers` package:

```
library(simhelpers)
create_skeleton()
```

This function will open up a new R script for you contains a template for a simulation study, with sections corresponding to each component.

3.5 Further readings & resources

- Morris, White, & Crowther (2019). Using simulation studies to evaluate statistical methods.
  - High-level simulation design considerations.
  - Details about performance criteria calculations.
  - Stata-centric.
- SimDesign R package (Chalmers, 2019)
  - Tools for building generic simulation workflows.
  - Chalmers & Adkin (2019). Writing effective and reliable Monte Carlo simulations with the SimDesign package.
- DeclareDesign (Blair, Cooper, Coppock, & Humphreys)
  - Specialized suite of R packages for simulating research designs.

  - Design philosophy is very similar to "tidy" simulation approach.

- SimHelpers R package (Joshi & Pustejovsky, 2020)

  - Helper functions for calculating performance criteria.
  - Includes Monte Carlo standard errors.

Chapter 4

Case Study: Heteroskedastic ANOVA

To illustrate the process of programming a simulation, let's look at the simulations from Brown and Forsythe (1974). We use this case study as a reoccuring example in the following chapters.

Brown and Forsythe wanted to study methods for testing hypotheses in the following model: Consider a population consisting of g separate groups, with population means $\mu_1, ..., \mu_g$ and population variances $\sigma_1^2, ..., \sigma_g^2$ for some characteristic X. We obtain samples of size $n_1, ..., n_g$ from each of the groups, and take measurements of the characteristic for each unit in each group. Let x_{ij} denote the measurement from unit j in group i, for $i = 1, ..., g$ and $j = 1, ..., n_i$. Our goal is to use the sample data to test the hypothesis that the population means are all equal, i.e.,

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_g.$$

Note that if the population variances were all equal (i.e., $\sigma_1^2 = \sigma_2^2 = \cdots = \sigma_g^2$), we could use a conventional one-way analysis of variance (ANOVA) to test. However, one-way ANOVA might not work well if the variances are not equal. The question is then what are best practices for testing, when one is in this heteroskedastic case.

To tackle this question, Brown and Forsythe evaluated two different hypothesis testing procedures, developed by James (1951) and Welch (1951), that had been proposed for testing this hypothesis without assuming equality of variances, along with the conventional one-way ANOVA F-test as a benchmark. They also proposed and evaluated a new procedure of their own devising. (This latter pieces makes this paper one of a canonical format for statistical methodology papers: find some problem that current procedures do not perfectly solve, invent something to do a better job, and then do simulation and/or math to build a case

that the new procedure is better.) Overall, the simulation involves comparing the performance of these different hypothesis testing procedures (the methods) under a range of conditions (different data generating processes).

For hypothesis testing, there are two main performance metrics of interest: type-I error rate and power. The type-I error rate is, when the null hypothesis is true, how often a test falsely rejects the null. It is a measure of how *valid* a method is. Power is how often a test correctly rejects the null when it is indeed false. It is a measure of how *powerful* or sensitive a method is. They explored error rates and power for nominal α-levels of 1%, 5%, and 10%. Table 1 of their paper reports the simulation results for type-I error (labeled as "size"); ideally, a test should have true type-I error very close to the nominal α. Table 2 reports results on power; it is desirable to have higher power to reject null hypotheses that are false, so higher rates are better here.

To replicate this simulation we are going to first write code to do a specific scenario with a specific set of core parameters (e.g., sample sizes, number of groups, and so forth), and then scale up to do a range of scenarios where we vary these parameters.

4.1 The data-generating model

In the heteroskedastic one-way ANOVA simulation, there are three sets of parameter values: population means, population variances, and sample sizes. Rather than attempting to write a general data-generating function immediately, it is often easier to write code for a specific case first and then use that code as a launch point for the rest. For example, say that we have four groups with means of 1, 2, 5, 6; variances of 3, 2, 5, 1; and sample sizes of 3, 6, 2, and 4:

```
mu <- c(1, 2, 5, 6)
sigma_sq <- c(3, 2, 5, 1)
sample_size <- c(3, 6, 2, 4)
```

Following Brown and Forsythe, we'll assume that the measurements are normally distributed within each sub-group of the population. The following code generates a vector of group id's and a vector of simulated measurements:

```
N <- sum(sample_size) # total sample size
g <- length(sample_size) # number of groups

# group id
group <- rep(1:g, times = sample_size)
```

```
# mean for each unit of the sample
mu_long <- rep(mu, times = sample_size)

# sd for each unit of the sample
sigma_long <- rep(sqrt(sigma_sq), times = sample_size)

# See what we have?
tibble( group=group, mu=mu_long, sigma=sigma_long)
```

```
## # A tibble: 15 x 3
##    group    mu sigma
##    <int> <dbl> <dbl>
##  1     1     1  1.73
##  2     1     1  1.73
##  3     1     1  1.73
##  4     2     2  1.41
##  5     2     2  1.41
##  6     2     2  1.41
##  7     2     2  1.41
##  8     2     2  1.41
##  9     2     2  1.41
## 10     3     5  2.24
## 11     3     5  2.24
## 12     4     6  1
## 13     4     6  1
## 14     4     6  1
## 15     4     6  1
```

```
# Now make our data
x <- rnorm(N, mean = mu_long, sd = sigma_long)
tibble(group = group, x = x)
```

```
## # A tibble: 15 x 2
##    group      x
##    <int>  <dbl>
##  1     1 -0.842
##  2     1  1.14
##  3     1  2.31
##  4     2  1.40
##  5     2  1.92
##  6     2  2.54
##  7     2 -1.09
##  8     2  4.47
##  9     2 -0.658
```

```
## 10     3  6.39
## 11     3  5.72
## 12     4  5.25
## 13     4  5.02
## 14     4  4.66
## 15     4  6.83
```

We have made a small dataset of group membership and outcome. We note that there are many different and legitimate ways of doing this in R. E.g., we could generate each group separately, and then stack our groups instead of using `rep` to do it all at once. In general, we advocate the adage that if you can do it at all, then you should feel good about yourself. I.e., don't worry about writing code the "best" way when you are initially putting a simulation together.

To continue, as we are going to generate data over and over, we wrap this code in a function. We also make our means, variances and sample sizes be parameters of our function so we can make datasets of different sizes and shapes, like so:

```
generate_data <- function(mu, sigma_sq, sample_size) {

  N <- sum(sample_size)
  g <- length(sample_size)

  group <- rep(1:g, times = sample_size)
  mu_long <- rep(mu, times = sample_size)
  sigma_long <- rep(sqrt(sigma_sq), times = sample_size)

  x <- rnorm(N, mean = mu_long, sd = sigma_long)
  sim_data <- tibble(group = group, x = x)

  return(sim_data)
}

sim_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                          sample_size = sample_size)
sim_data
```

```
## # A tibble: 15 x 2
##    group     x
##    <int> <dbl>
##  1     1 2.32
##  2     1 2.57
##  3     1 0.534
##  4     2 0.373
##  5     2 2.32
##  6     2 0.155
```

```
##  7     2 2.88
##  8     2 0.733
##  9     2 1.99
## 10     3 8.23
## 11     3 6.62
## 12     4 5.96
## 13     4 6.65
## 14     4 3.70
## 15     4 6.34
```

The above code is just the code we built previously, all bundled up. Our workflow is to scrabble around to get it to work once, the way we want, and then bundle up our final work into a function for later reuse.

Each time we run the function we get a new set of simulated data:

```
generate_data(mu = mu, sigma_sq = sigma_sq, sample_size = sample_size)
```

```
## # A tibble: 15 x 2
##    group      x
##    <int>  <dbl>
##  1     1  1.19
##  2     1 -0.298
##  3     1  1.31
##  4     2  2.62
##  5     2 -3.19
##  6     2  3.13
##  7     2  2.24
##  8     2  2.99
##  9     2  1.70
## 10     3  0.832
## 11     3  6.28
## 12     4  6.25
## 13     4  6.11
## 14     4  2.04
## 15     4  7.27
```

4.2 The estimation procedures

Brown and Forsythe considered four different hypothesis testing procedures for heteroskedastic ANOVA. For starters, let's look at the simplest one, which is just to use a conventional one-way ANOVA (while mistakenly assuming homoskedasticity). R's `oneway.test` function will actually calculate this test automatically:

```
sim_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                          sample_size = sample_size)
oneway.test(x ~ factor(group), data = sim_data, var.equal = TRUE)
```

```
##
##  One-way analysis of means
##
## data:  x and factor(group)
## F = 8.0596, num df = 3, denom df = 11, p-value = 0.004044
```

The main result we need here is the p-value, which will let us assess the test's Type-I error and power for a given nominal α-level. The following function takes simulated data as input and returns as output the p-value from a one-way ANOVA:

```
ANOVA_F_aov <- function(sim_data) {
  oneway_anova <- oneway.test(x ~ factor(group), data = sim_data,
                              var.equal = TRUE)
  return(oneway_anova$p.value)
}

ANOVA_F_aov(sim_data)
```

```
## [1] 0.004043699
```

An alternative approach would be to "hand roll" the ANOVA F statistic and test directly. Doing so by hand can set you up to implement modified versions of these tests later on. Also, although hand-building a method does take more work to program, it can result in a faster piece of code (this actually is the case here) which in turn can make the overall simulation faster.

Following the formulas on p. 129 of Brown and Forsythe (1974) we have:

```
ANOVA_F <- function(sim_data) {

  x_bar <- with(sim_data, tapply(x, group, mean))
  s_sq <- with(sim_data, tapply(x, group, var))
  n <- table(sim_data$group)
  g <- length(x_bar)

  df1 <- g - 1
  df2 <- sum(n) - g

  msbtw <- sum(n * (x_bar - mean(sim_data$x))^2) / df1
```

```
  mswn <- sum((n - 1) * s_sq) / df2
  fstat <- msbtw / mswn
  pval <- pf(fstat, df1, df2, lower.tail = FALSE)

  return(pval)
}

ANOVA_F(sim_data)
```

```
## [1] 0.004043699
```

To see the difference between our version and R's version, we can use an R package called `microbenchmark` to test how long the computations take for each version of the function. The `microbenchmark` function runs each expression 100 times (by default) and tracks how long the computations take. It then summarizes the distribution of timings:

```
library(microbenchmark)
timings <- microbenchmark(Rfunction = ANOVA_F_aov(sim_data),
                          direct    = ANOVA_F(sim_data))
timings
```

```
## Unit: microseconds
##       expr     min       lq     mean  median        uq       max neval
##  Rfunction 693.402 746.8005 1165.869 809.501 1207.9010 10012.202   100
##     direct 293.901 318.0510  485.218 342.000  503.8005  3872.401   100
```

The direct function is 2.4 times faster than the built-in R function.

This result is not unusual. Built-in R functions usually include lots of checks and error-handling, which take time to compute. These checks are crucial for messy, real-world data analysis but unnecessary with our pristine, simulated data. Here we can skip them by doing the calculations directly. In general, however, this is a trade-off: writing something yourself gives you a lot of chance to do something wrong, throwing off all your simulations. It might be faster, but you may pay dearly for it in terms of extra hours coding and debugging. Optimize only if you need to!

Now let's consider the Welch test, another one of the tests considered by Brown and Forsythe. Here is a function that calculates the Welch test, again following the notation and formulas from the paper:

```
Welch_F <- function(sim_data) {
```

```
  x_bar <- with(sim_data, tapply(x, group, mean))
  s_sq <- with(sim_data, tapply(x, group, var))
  n <- table(sim_data$group)
  g <- length(x_bar)

  w <- n / s_sq
  u <- sum(w)
  x_tilde <- sum(w * x_bar) / u
  msbtw <- sum(w * (x_bar - x_tilde)^2) / (g - 1)

  G <- sum((1 - w / u)^2 / (n - 1))
  denom <- 1 +  G * 2 * (g - 2) / (g^2 - 1)
  W <- msbtw / denom
  f <- (g^2 - 1) / (3 * G)

  pval <- pf(W, df1 = g - 1, df2 = f, lower.tail = FALSE)

  return(pval)
}

Welch_F(sim_data)
```

```
## [1] 0.04428594
```

4.3 Replication

We now have functions that implement steps 2 and 3 of the simulation. Given some parameters, `generate_data` produces a simulated dataset and `ANOVA_F` and `Welch_F` use the simulated data to calculate p-values two different ways. We now want to know which way is better, and how. To answer this question, we next need to repeat this chain of calculations a bunch of times.

We first make a function that puts our chain together in a single method. This method is also responsible for putting the results together in a tidy structure that is easy to use.

```
one_run = function( mu, sigma_sq, sample_size ) {
  sim_data <- generate_data(mu = mu, sigma_sq = sigma_sq, sample_size = sample_size)
  anova_p <- ANOVA_F(sim_data)
  Welch_p <- Welch_F(sim_data)
  tibble(ANOVA = anova_p, Welch = Welch_p)
}

one_run( mu = mu, sigma_sq = sigma_sq, sample_size = sample_size )
```

```
## # A tibble: 1 x 2
##       ANOVA   Welch
##       <dbl>   <dbl>
## 1 0.0000564 0.00881
```

A single simulation run should do steps 2 and 3, ending with a nice dataframe or tibble that has our results for that single run.

We next do this over and over. There are several ways to do this. We have seen `rerun()` before. Another, more classic, way is to use an R function called `replicate`; `rerun()` and `replicate()` are near equivilents. `replicate()` does what its name suggests—it replicates the result of an expression a specified number of times. The first argument is the number of times to replicate and the next argument is an expression (a short piece of code to run). A further argument, `simplify` allows you to control how the results are structured. Setting `simplify = FALSE` returns the output as a list. The following code produces four replications of our simulation:

```
sim_data <- replicate(n = 4,
                      one_run(mu = mu, sigma_sq = sigma_sq,
                              sample_size = sample_size),
                      simplify = FALSE)
str(sim_data)
```

```
## List of 4
##  $ : tibble [1 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ ANOVA: num 0.00754
##   ..$ Welch: num 0.07
##  $ : tibble [1 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ ANOVA: num 0.00237
##   ..$ Welch: num 0.052
##  $ : tibble [1 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ ANOVA: num 0.0127
##   ..$ Welch: num 0.0416
##  $ : tibble [1 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ ANOVA: num 0.00475
##   ..$ Welch: num 0.0303
```

The `bind_rows` function from the `dplyr` package will stack all of the data frames in our list into a single data frame, for easier manipulation:

```
library(dplyr)
bind_rows(sim_data)
```

```
## # A tibble: 4 x 2
```

```
##     ANOVA  Welch
##     <dbl>  <dbl>
## 1 0.00754 0.0700
## 2 0.00237 0.0520
## 3 0.0127  0.0416
## 4 0.00475 0.0303
```

Voila! Simulated p-values!

4.4 Calculating rejection rates

We've got all the pieces in place now to reproduce the results from Brown and Forsythe (1974). Let's focus on calculating the actual type-I error rate of these tests—that is, the proportion of the time that they reject the null hypothesis of equal means when that null is actually true—for an α-level of .05. We therefore need to simulate data according to process where the population means are indeed all equal. Arbitrarily, let's look at $g = 4$ groups and set all of the means equal to zero:

```
mu <- rep(0, 4)
```

In the fifth row of Table 1, Brown and Forsythe examine performance for the following parameter values for sample size and population variance:

```
sample_size <- c(4, 8, 10, 12)
sigma_sq <- c(3, 2, 2, 1)^2
```

With these parameter values, we can use our `replicate` code to simulate 10,000 p-values:

```
p_vals <- replicate(n = 10000,
  sim_data <- one_run(mu = mu, sigma_sq = sigma_sq, sample_size = sample_size),
  simplify = FALSE)

p_vals <- bind_rows(p_vals)
p_vals
```

```
## # A tibble: 10,000 x 2
##     ANOVA Welch
##     <dbl> <dbl>
##  1 0.912  0.940
##  2 0.0829 0.145
```

```
##  3 0.213  0.412
##  4 0.677  0.617
##  5 0.916  0.934
##  6 0.938  0.907
##  7 0.876  0.911
##  8 0.944  0.893
##  9 0.0795 0.196
## 10 0.808  0.868
## # ... with 9,990 more rows
```

Now how to calculate the rejection rates? The rule is that the null is rejected if the p-value is less than α. To get the rejection rate, calculate the proportion of replications where the null is rejected. This is equivalent to taking the mean of the logical conditions:

```
mean(p_vals$ANOVA < 0.05)
```

```
## [1] 0.1409
```

We get a rejection rate that is much larger than $\alpha = .05$, which indicates that the ANOVA F-test does not adequately control Type-I error under this set of conditions.

```
mean(p_vals$Welch < 0.05)
```

```
## [1] 0.0658
```

The Welch test does much better, although it is still a little bit in excess of 0.05.

Note that these two numbers are quite close (though not quite identical) to the corresponding entries in Table 1 of Brown and Forsythe (1974). The difference is due to the fact that both Table 1 and are results are actually *estimated* rejection rates, because we haven't actually simulated an infinite number of replications. The estimation error arising from using a finite number of replications is called *simulation error* (or *Monte Carlo error*). Later on, we'll look more at how to estimate and control the simulation error in our studies.

4.5 Exercises

The following exercises involve exploring and tweaking the simulation code we've developed to replicate the results of Brown and Forsythe (1974) that we provide in the above.

1. Table 1 from Brown and Forsythe reported rejection rates for $\alpha = .01$ and $\alpha = .10$ in addition to $\alpha = .05$. Calculate the rejection rates of the ANOVA F and Welch tests for all three α-levels.

2. Try simulating the Type-I error rates for the parameter values in the first two rows of Table 1 of the original paper. Use 10,000 replications. How do your results compare to the results reported in the Table?

3. Try simulating the **power levels** for a couple of sets of parameter values from Table 2. Use 10,000 replications. How do your results compare to the results reported in the Table?

4. One might instead of having `one_run` return a single row with the columns for the p-values, have multiple rows with each row being a test (so one row for ANOVA and one for Welch). Modify `one_run()` to do this, and then use `group_by()` plus `summarise()` to calculate rejection rates in one go. This might be nicer if we had more than two methods, or if each method returned not just a p-value but other quantaties of interest.

Chapter 5

Data-generating models

The data generating model, or data generating process (DGP), is the recipe we use to create fake data that we will use for analysis. When we generate from a model we know what the "right answer" is. We can then compare our estimates to this right answer, to assess whether our estimation procedures worked.

The easiest way to describe a DGP is usually via a sequence of equations and random variables. We then convert these equations to code by following the steps laid out.

For example, for the Welch data earlier we might have, for observation i in group g:

$$X_{ig} = \mu_g + \epsilon_{ig} \text{ with } \epsilon_{ig} \sim N(0, \sigma_g^2)$$

These math equations would also come along with specified parameter values (the μ_g, etc), and sample size requirements.

In terms of code, a function that implements a data-generating model should have the following form:

```
generate_data <- function(parameters) {

  # generate pseudo-random numbers and use those to
  # make some data

  return(sim_data)
}
```

The function takes a set of parameter values as input, simulates random numbers and does calculations, and produces as output a set of simulated data. Again, there will in general be multiple parameters, and these will include not only the

model parameters (e.g. the coefficients of a regression), but also sample sizes and other study design parameters. The output will typically be a dataframe, mimicking what data one would see in the "real world," possibly augmented by some other latent values that we can use later on to assess whether the estimation procedures we are checking are close to the truth.

For example, from our Welch case study, we have the following method that generates grouped data with a single outcome.

```
generate_data <- function(mu, sigma_sq, sample_size) {

  N <- sum(sample_size)
  g <- length(sample_size)

  group <- rep(1:g, times = sample_size)
  mu_long <- rep(mu, times = sample_size)
  sigma_long <- rep(sqrt(sigma_sq), times = sample_size)

  x <- rnorm(N, mean = mu_long, sd = sigma_long)
  sim_data <- tibble(group = group, x = x)

  return(sim_data)
}

mu <- c(1, 2, 5, 6)
sigma_sq <- c(3, 2, 5, 1)
sample_size <- c(3, 6, 2, 4)

sim_dat <- generate_data(mu = mu,
                         sigma_sq = sigma_sq,
                         sample_size = sample_size)
sim_dat
```

```
## # A tibble: 15 x 2
##    group       x
##    <int>   <dbl>
##  1     1 -0.599
##  2     1  0.0135
##  3     1  2.82
##  4     2  2.92
##  5     2  1.52
##  6     2  3.67
##  7     2  2.00
##  8     2  1.19
##  9     2  2.22
## 10     3  1.55
```

```
## 11     3  9.32
## 12     4  5.63
## 13     4  6.72
## 14     4  3.43
## 15     4  6.69
```

5.1 Computational efficiency versus simplicity

An alternative approach to the above would be to write a function that generates *multiple* sets of simulated data all at once. For example, we could specify that we want `R` replications of the study and have the function spit out a matrix with `R` columns, one for each simulated dataset:

```
generate_data_matrix <- function(mu, sigma_sq, sample_size, R) {

  N <- sum(sample_size)
  g <- length(sample_size)

  group <- rep(1:g, times = sample_size)
  mu_long <- rep(mu, times = sample_size)
  sigma_long <- rep(sqrt(sigma_sq), times = sample_size)

  x_mat <- matrix(rnorm(N * R, mean = mu_long, sd = sigma_long),
                  nrow = N, ncol = R)
  sim_data <- list(group = group, x_mat = x_mat)

  return(sim_data)
}

generate_data_matrix(mu = mu, sigma_sq = sigma_sq,
                     sample_size = sample_size, R = 4)
```

```
## $group
##  [1] 1 1 1 2 2 2 2 2 2 3 3 4 4 4 4
##
## $x_mat
##            [,1]        [,2]       [,3]       [,4]
##  [1,] 0.6981179 -1.77908576  0.9707895 0.09497638
##  [2,] 4.5754748  5.48641785  3.6732408 4.68991858
##  [3,] 0.1891774  3.14554439  1.5998286 3.36306663
##  [4,] 0.5026508  3.06185620 -0.1989293 3.99729213
##  [5,] 1.4981045  0.06876396  0.8030075 1.52567632
##  [6,] 3.8668794  2.31238700  3.1337866 3.27995096
##  [7,] 2.1067668  3.54368722  3.7036162 2.68014322
```

```
##  [8,] 3.1456969  3.88446390  2.1655479 3.16865004
##  [9,] 2.3356898  1.98563979  2.7872473 1.97506997
## [10,] 2.7192518  1.73069858  8.2006790 4.18723326
## [11,] 2.6963880  2.73046740  4.5045510 5.58514955
## [12,] 6.2748546  5.66172582  7.1782574 5.85854862
## [13,] 6.9068274  5.17548577  7.4077842 7.01033667
## [14,] 5.9421043  8.01215429  5.9639092 7.61052227
## [15,] 5.9484805  5.48181200  6.0351882 5.53799805
```

This approach is a bit more computationally efficient because the setup calculations (getting `N`, `g`, `group`, `mu_full`, and `sigma_full`) only have to be done once instead of once per replication. It also makes clever use of vector recycling in the call to `rnorm()`. However, the structure of the resulting data is more complicated, which will make it more difficult to do the later estimation steps. Furthermore, if the number of replicates `R` is large and each replication produces a large dataset, this "all-at-once" approach will entail generating and holding very large amounts of data in memory, which can create other performance issues. On balance, we recommend following the simpler approach of writing a function that generates a single simulated dataset per call (unless and until you have a principled reason to do otherwise).

Beyond a few obvious coding tricks we will discuss, one should optimize code only after you discover you need to. Optimizing as you go usually means you will spend a lot of time wrestling with code far more complicated than it needs to be. For example, often it is the estimation method that will take a lot of computational time, so having very fast data generation code won't help overall simulation runtimes much, as you are tweaking something that is only a small part of the overall pie, in terms of time. Keep things simple; your time is more important than the computer's time.

5.2 Checking the data-generating function

An important part of programing in R—particularly writing functions—is finding ways to test and check the correctness of your code. Thus, after writing a data-generating function, we need to consider how to test whether the output it produces is correct. How best to do this will depend on the data-generating model being implemented.

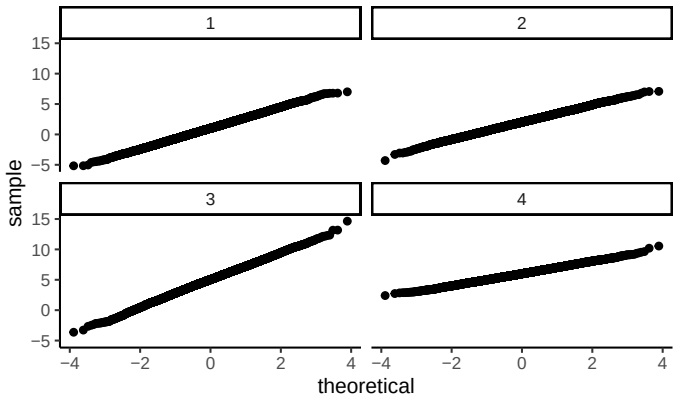
For the heteroskedastic ANOVA problem, one basic thing we could do is check that the simulated data from each group follows a normal distribution. By generating very large samples from each group, we can effectively check characteristics of the population distribution. In the following code, we simulate very large samples from each of the four groups, and check that the means and variances agree with the input parameters:

```
check_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                            sample_size = rep(10000, 4))

chk <- check_data %>% group_by( group ) %>%
  summarise( n = n(),
             mean = mean( x ),
             var = var( x ) ) %>%
  mutate( mu = mu,
          sigma2 = sigma_sq ) %>%
  relocate( group, n, mean, mu, var, sigma2 )
chk
```

```
## # A tibble: 4 x 6
##   group     n  mean    mu   var sigma2
##   <int> <int> <dbl> <dbl> <dbl>  <dbl>
## 1     1 10000  1.02     1  2.99      3
## 2     2 10000  2.01     2  2.02      2
## 3     3 10000  5.02     5  4.98      5
## 4     4 10000  6.01     6  1.03      1
```

We can also make some diagnostic plots to assess whether we have normal data (using QQ plots, where we expect a straight line if the data are normal):

```
ggplot( check_data, aes( sample=x ) ) +
  facet_wrap( ~ group ) +
  stat_qq()
```



5.3 Exercises

5.3.1 Shifted-and-scaled t distribution

The shifted-and-scaled t-distribution has parameters μ (mean), σ (scale), and ν (degrees of freedom). If T follows a student's t-distribution with ν degrees of freedom, then $S = \mu + \sigma T$ follows a shifted-and-scaled t-distribution. The following function will generate random draws from this distribution (the scaling of $(\nu - 2)/\nu$ is to account for a non-scaled t-distribution having a variance of $\nu/(\nu - 2)$).

```
r_tss <- function(n, mean, sd, df) {
  mean + sd * sqrt( (df-2)/df ) * rt(n = n, df = df)
}

r_tss(n = 8, mean = 3, sd = 2, df = 5)
```

```
## [1]  4.1854338 -0.2148387  7.0402290  4.9054148  3.2893721  4.9844551  5.1223279
## [8]  1.7346387
```

Modify that `simulate_data` function to generate data from shifted-and-scaled t-distributions rather than from normal distributions. Include the degrees of freedom as an input argument. Re-run the Type-I error rate calculations from the prior exercises with a t-distribution with 5 degrees of freedom. Do the results change substantially?

Chapter 6

Estimation procedures

In the abstract, a function that implements an estimation procedure should have the following form:

```
estimate <- function(sim_data) {

  # calculations/model-fitting/estimation procedures

  return(estimates)
}
```

The function takes a set of data as input, fits a model or otherwise calculates an estimate, possibly with associated standard errors and so forth, and produces as output these estimates. In principle, you should be able to run your function on real data as well as simulated.

The estimates could be point-estimates of parameters, standard errors, confidence intervals, etc. Depending on the research question, this function might involve a combination of several procedures (e.g., a diagnostic test for heteroskedasticity, followed by the conventional formula or heteroskedasticity-robust formula for standard errors). Also depending on the research question, we might need to create *several* functions that implement different estimation procedures to be compared.

In Chapter @ref(case_ANOVA), for example, we saw different functions for some of the methods Brown and Forsythe considered for heteroskedastic ANOVA. We re-print them here, taking full advantage of our digital-bookness:

```
ANOVA_F <- function(sim_data) {

  x_bar <- with(sim_data, tapply(x, group, mean))
```

```
  s_sq <- with(sim_data, tapply(x, group, var))
  n <- table(sim_data$group)
  g <- length(x_bar)

  df1 <- g - 1
  df2 <- sum(n) - g

  msbtw <- sum(n * (x_bar - mean(sim_data$x))^2) / df1
  mswn <- sum((n - 1) * s_sq) / df2
  fstat <- msbtw / mswn
  pval <- pf(fstat, df1, df2, lower.tail = FALSE)

  return(pval)
}


Welch_F <- function(sim_data) {

  x_bar <- with(sim_data, tapply(x, group, mean))
  s_sq <- with(sim_data, tapply(x, group, var))
  n <- table(sim_data$group)
  g <- length(x_bar)

  w <- n / s_sq
  u <- sum(w)
  x_tilde <- sum(w * x_bar) / u
  msbtw <- sum(w * (x_bar - x_tilde)^2) / (g - 1)

  G <- sum((1 - w / u)^2 / (n - 1))
  denom <- 1 +  G * 2 * (g - 2) / (g^2 - 1)
  W <- msbtw / denom
  f <- (g^2 - 1) / (3 * G)

  pval <- pf(W, df1 = g - 1, df2 = f, lower.tail = FALSE)

  return(pval)
}
```

6.1 Further notes on computational efficiency

Computational and programming efficiency is usually a secondary consideration when you're starting to design a simulation study. It's better to produce accurate code, even if it's a bit slow, than to write code that is speedy but hard to follow (or even worse, that produces incorrect results). All that said, there

is some glaring redundancy in the two functions above. Both of them start by taking the simulated data and calculating summary statistics for each group, using the following code:

```
x_bar <- with(sim_data, tapply(x, group, mean))
s_sq <- with(sim_data, tapply(x, group, var))
n <- table(sim_data$group)
g <- length(x_bar)
```

In the interest of not repeating ourselves, it would better to pull this code out as a separate function and then re-write the `ANOVA_F` and `Welch_F` functions to take the summary statistics as input. Here is a function that takes simulated data and returns a list of summary statistics:

```
summarize_data <- function(sim_data) {

  res <- sim_data %>% group_by( group ) %>%
    summarise( x_bar = mean( x ),
               s_sq = var( x ),
               n = n() )
  res
}
```

We just packaged the code from above, and puts our results in a nice table (and thus pivoted to using tidyverse to calculate these things):

```
sim_data = generate_data(mu=mu, sigma_sq=sigma_sq, sample_size=sample_size)
summarize_data(sim_data)
```

```
## # A tibble: 4 x 4
##   group  x_bar  s_sq     n
##   <int>  <dbl> <dbl> <int>
## 1     1 -0.959 1.92      3
## 2     2  2.05  0.611     6
## 3     3  5.07  0.986     2
## 4     4  6.20  0.964     4
```

Now we can re-write both F-test functions to use the output of this function:

```
ANOVA_F_agg <- function(x_bar, s_sq, n) {
  g = length(x_bar)
  df1 <- g - 1
  df2 <- sum(n) - g
```

```
  msbtw <- sum(n * (x_bar - weighted.mean(x_bar, w = n))^2) / df1
  mswn <- sum((n - 1) * s_sq) / df2
  fstat <- msbtw / mswn
  pval <- pf(fstat, df1, df2, lower.tail = FALSE)

  return(pval)
}

summary_stats <- summarize_data(sim_data)
with(summary_stats, ANOVA_F_agg(x_bar = x_bar, s_sq = s_sq, n = n))
```

```
## [1] 6.671062e-06
```

```
Welch_F_agg <- function(x_bar, s_sq, n) {
  g = length(x_bar)
  w <- n / s_sq
  u <- sum(w)
  x_tilde <- sum(w * x_bar) / u
  msbtw <- sum(w * (x_bar - x_tilde)^2) / (g - 1)

  G <- sum((1 - w / u)^2 / (n - 1))
  denom <- 1 +  G * 2 * (g - 2) / (g^2 - 1)
  W <- msbtw / denom
  f <- (g^2 - 1) / (3 * G)

  pval <- pf(W, df1 = g - 1, df2 = f, lower.tail = FALSE)

  return(pval)
}

with(summary_stats, ANOVA_F_agg(x_bar = x_bar, s_sq = s_sq, n = n))
```

```
## [1] 6.671062e-06
```

The results are the same as before.

We then put all these pieces in our revised `one_run()` method as so:

```
one_run_fast <- function( mu, sigma_sq, sample_size ) {
  sim_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                            sample_size = sample_size)
  summary_stats <- summarize_data(sim_data)
  anova_p <- with(summary_stats,
                  ANOVA_F_agg(x_bar = x_bar,s_sq = s_sq, n = n))
```

```
  Welch_p <- with(summary_stats,
                  Welch_F_agg(x_bar = x_bar, s_sq = s_sq, n = n))
  tibble(ANOVA = anova_p, Welch = Welch_p)
}

one_run_fast( mu = mu, sigma_sq = sigma_sq,
              sample_size = sample_size )
```

```
## # A tibble: 1 x 2
##    ANOVA Welch
##    <dbl> <dbl>
## 1 0.0209 0.107
```

The reason this is important is we are now doing our group aggregation only once, rather than once per method. We can use our microbenchmark to see our speedup:

```
library(microbenchmark)
timings <- microbenchmark(noagg = one_run(mu = mu, sigma_sq = sigma_sq,
                                          sample_size = sample_size),
                          agg = one_run_fast(mu = mu, sigma_sq = sigma_sq,
                                             sample_size = sample_size) )
timings
```

```
## Unit: milliseconds
##   expr      min       lq     mean   median       uq      max neval
##  noagg 1.258401 1.370301 2.023650 1.450201 1.595501  40.2360   100
##    agg 4.199502 4.791100 6.319656 4.941251 5.339201 108.1927   100
```

And our relative speedup is:

```
with(summary(timings), round(mean[1] / mean[2], 1))
```

```
## [1] 0.3
```

To recap, there are two advantages of this kind of coding:

1. Code reuse is generally good because when you have the same code in multiple places it can make it harder to read and understand your code. If you see two blocks of code you might worry they are only mostly similar, not exactly similar, and waste time trying to differentiate. If you have a single, well-named function, you immediately know what a block of code is doing.

2. Saving the results of calculations can speed up your computation since you are saving your partial work. This can be useful to reduce calculations that are particularly time intensive.

6.2 Checking the estimation function

Just as with the data-generating function, it is important to verify the accuracy of the estimation functions. For the ANOVA-F test, this can be done simply by checking the result of our `ANOVA_F` against the built-in `oneway.test` function. Let's do that with a fresh set of data:

```
sim_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                          sample_size = sample_size)
aov_results <- oneway.test(x ~ factor(group), data = sim_data,
                           var.equal = TRUE)
aov_results
```

```
##
##  One-way analysis of means
##
## data:  x and factor(group)
## F = 5.8048, num df = 3, denom df = 11, p-value = 0.01251
```

```
summary_stats <- summarize_data(sim_data)
F_results <- with(summary_stats,
                  ANOVA_F_agg(x_bar = x_bar, s_sq = s_sq, n = n))
F_results
```

```
## [1] 0.01251098
```

```
all.equal(aov_results$p.value, F_results)
```

```
## [1] TRUE
```

We use `all.equal()` because it will check equality up to a tolerance in R, which can avoid some weird floating point errors due to rounding in R.

We can follow the same approach to check the results of the Welch test because it is also implemented in `oneway.test`:

```
aov_results <- oneway.test(x ~ factor(group), data = sim_data, var.equal = FALSE)
aov_results
```

```
##
##  One-way analysis of means (not assuming equal variances)
##
## data:  x and factor(group)
## F = 6.3506, num df = 3.0000, denom df = 3.1795, p-value = 0.07507
```

```
W_results <- with(summary_stats, Welch_F_agg(x_bar = x_bar, s_sq = s_sq, n = n))
W_results
```

```
## [1] 0.07507405
```

```
all.equal(aov_results$p.value, W_results)
```

```
## [1] TRUE
```

Checking estimation functions can be a bit more difficult for procedures that are not already implemented in R. For example, the two other procedures examined by Brown and Forsythe, the James' test and Brown and Forsythe's $F*$ test, are not available in base R. They are, however, available in the user-contributed package `onewaytests` (I found this by searching for "Brown-Forsythe" at http://rseek.org/). We could benchmark our calculations against this package, but of course there is some risk that the package might not be correct. Another route is to verify your results on numerical examples reported in authoritative papers, on the assumption that there's less risk of an error there. In the original paper that proposed the test, Welch (1951) provides a worked numerical example of the procedure. He reports the following summary statistics:

```
g <- 3
x_bar <- c(27.8, 24.1, 22.2)
s_sq <- c(60.1, 6.3, 15.4)
n <- c(20, 20, 10)
```

He also reports $W = 3.35$ and $f = 22.6$. Replicating the calculations with our `Welch_F` function:

```
Welch_F_agg(x_bar = x_bar, s_sq = s_sq, n = n)
```

```
## [1] 0.05479049
```

We get slightly different results! But we know that our function is correct—or at least consistent with `oneway.test`—so what's going on? It turns out that there was an error in some of Welch's intermediate calculations, which can only be spotted because he reported all of his work in the paper.

6.3 Exercises

The following exercises involve exploring and tweaking the simulation code we've developed to replicate the results of Brown and Forsythe (1974). Below is the key functions for the data-generating process (taken from the prior chapter). The estimation procedures are above.

```
generate_data <- function(mu, sigma_sq, sample_size) {

  N <- sum(sample_size)
  g <- length(sample_size)

  group <- rep(1:g, times = sample_size)
  mu_long <- rep(mu, times = sample_size)
  sigma_long <- rep(sqrt(sigma_sq), times = sample_size)

  x <- rnorm(N, mean = mu_long, sd = sigma_long)
  sim_data <- data.frame(group = group, x = x)

  return(sim_data)
}
```

6.3.1 Adding the BFF* test

Write a function that implements the Brown-Forsythe F*-test (the BFF* test!) as described on p. 130 of Brown and Forsythe (1974). Incorporate the function into the `one_run()` function from the previous question, and use it to estimate rejection rates of the BFF* test for the parameter values in the fifth line of Table 1 (which are the same as those used in the previous question).

```
BF_F <- function(x_bar, s_sq, n, g) {

  # fill in the guts here

  return(pval = pval)
}

# Further R code here
```

Chapter 7

Performance criteria

So far, we've looked at the structure of simulation studies and seen how to write functions that generate data according to a specified model (and parameters) and functions that implement estimation procedures on simulated data. Put those two together and repeat a bunch of times, and we'll have a lot of estimates and perhaps also their estimated standard errors and/or confidence intervals. And if the purpose of the simulation is to compare *multiple* estimation procedures, then we'll have a set of estimates (SEs, CIs, etc.) for *each* of the procedures. The question is then: how do we assess the performance of these estimators?

In this chapter, we'll look at a variety of **performance criteria** that are commonly used to compare the relative performance of multiple estimators or measure how well an estimator works. These performance criteria are all assessments of how the estimator behaves if you repeat the experimental process an infinite number of times. In statistical terms, these criteria are summaries of the true sampling distribution of the estimator, given a specified data generating process.

Although we can't observe this sampling distribution directly (and it can only rarely be worked out in full mathematical detail), we can *sample* from it. In particular, the set of estimates generated from a simulation constitute a (typically large) sample from the sampling distribution of an estimator. (Say that six times fast!) We then use that sample to *estimate* the performance criteria of interest. For example, we want to know what percent of the time we would reject the null hypothesis; we estimate this by seeing how often we do in 1000 trials. Now, because we have only a sample of trials rather than the full distribution, our estimates are estimates. In other words, they can be wrong, just due to random chance. We can describe how wrong with the **Monte Carlo standard error (MCSE)**; it is the standard error in our estimation of performance due to the simulation only having a finite number of trials. Just as with statistical uncertainty when analyzing data, we can estimate our MCSE and even generate confidence intervals for our performance estimates with them. And then, if we

can computationally do it, we would use a large enough number of replications so that the MCSE is small enough that our performance estimates have our desired level of precision.

7.1 Inference vs. Estimation

A core purpose of simulation is to compare different estimators to each other under a variety of circumstances. These sorts of simulations are simulations to examine and compare the properties of different inferential methods or estimators. For example, one might want to know how much better adjusting for covariates is, if at all, when analyzing a randomized experiment.

Such an analysis might involve two general activities: inference and estimation. *Inference* is when we do hypothesis testing, asking whether there is evidence for some sort of effect, or asking whether there is evidence that some coefficient is greater than or less than some specified value. So in our circumstance, we might specify interest in the average treatment effect, which we will call τ. Inference would be testing the null of $H_0 : \tau = 0$.

Estimation is when we estimate the actual value of τ. There might be different ways of obtaining some estimate, and we want to know which is best. Hand-in-hand with estimation is estimating uncertainty, i.e. assessing how close we believe our estimate to be to the truth. Here we would examine how well, for example, different estimators of the standard error perform.

For our hypothetical scenario, we might consider using either of two estimators of interest, the simple difference in means,

$$\hat{\tau}_{sd} = \overline{Y}_1 - \overline{Y}_0,$$

where $\overline{Y}_z$ is the average outcome of those units given treatment z, versus the coefficient $\hat{\tau}_{ols}$ from fitting the ordinary regression:

$$Y = a + bX + \tau Z + \epsilon.$$

Call these two Estimator A and Estimator B. For $\hat{\tau}_{sd}$ we would use Neyman's formula for estimating the standard error:

$$\widehat{SE}(\tau_{sd}) = \frac{\hat{\sigma}_1^2}{n_1} + \frac{\hat{\sigma}_0^2}{n_0}.$$

For $\hat{\tau}_{ols}$ we might use, say, the usual standard errors we get from generic statistics software. Finally, for inference, we would perhaps use the p-value from a Wald test for our simple difference in means and the p-values we get from the regression command.

The question would then be whether our two estimation strategies were different, whether one was superior to the other, and what salient differences were. In

particular, for our simple example, we might want to know if there is evidence that there is a treatment effect at all. This would be inference. We might also want to know what the average treatment effect is. This is estimation. These two goals are clearly highly related – if we have a good estimate of the treatment effect and it is not zero, then we are willing to say that there is a treatment effect – but depending on the framing, the way you would set up a simulation to investigate the behavior of your estimators will be different.

For inference, we first might ask whether both our methods are valid, i.e., ask whether these methods work correctly when we test for a treatment effect when there is none. In particular, we might wonder whether adjusting for a covariate could open the door to inference problems if there was no actual treatment effect, but where the residuals had some non-normal distribution. These sorts of questions are questions of validity.

Also for inference, we might ask which method is better for detecting an effect when there is one. Here, we want to know how these estimators perform in circumstances with a non-zero average treatment effect. Do they reject the null often, or rarely? How much does including covariates increase our chances of rejection? These are questions about power.

For estimation, we can be concerned with two things: bias and variance. An estimator is biased if it would generally give estimates that are higher (or lower) than the parameter being estimated. The variance of an estimator is how much the estimator varies from trial to trial. The variance is the true standard error, squared.

We might also be concerned with how well we can estimate the uncertainty of our estimators (i.e., estimate our standard error). For example, we might have an estimator that works very well, but we have no ability to estimate how well in any given circumstance.

7.2 Evaluation of Estimation Methods

Estimation has two major components, the point estimator and the uncertainty estimator. We evaluate both the *actual* properties of the point estimator and the performance of the *estimated* properties of the point estimator. For example, consider a specific estimate $\hat{\tau}$ of our average treatment effect. We first wish to know the actual bias and true standard error (SE) of $\hat{\tau}$. These are its actual properties. However, for each estimated $\hat{\tau}$, we also estimate $\widehat{SE}$, as our estimated measure of how precise our estimate is. We need to understand the properties of $\widehat{SE}$ as well.

7.2.1 Assessing actual properties

These are simple. For a given scenario, we repeatedly generate data and estimate effects. We then take the mean and standard deviation of these repeated trials to estimate actual properties via Monte Carlo. Given sufficient simulation trials, we can obtain arbitrarily accurate measures.

For example, we can ask what the *actual* variance (or standard error) of our estimator is. We can ask if our estimator is biased. We can ask what the overall $RMSE$ (root mean squared error) of our estimator is.

7.2.2 Assessing estimated properties

Let our estimator be $\hat{\tau}$. In our simulation we can know its actual properties, but if we were to use this estimator in practice we would have to also estimate its associated standard error, and generate confidence intervals and so forth. To understand if this works, we need to evaluate not only the behavior of the estimator itself, but the behavior of these associated things.

7.3 Assessing a point estimator

Consider an estimator T for a parameter θ. A simulation study generates a (typically large) sample of estimates $T_1, ..., T_R$, all of the target θ.

The most common measures of an estimator are the bias, variance, and mean squared error. We can first assess whether our estimator is biased, by comparing the mean of our R estimates

$$\bar{T} = \frac{1}{R}\sum_{r=1}^{R} T_r$$

to θ. The bias of our estimator is $bias = \bar{T} - \theta$.

We can also ask how variable our estimator is, by assessing the size of the variance of our R estimates

$$S_T^2 = \frac{1}{R-1}\sum_{r=1}^{R} \left(T_r - \bar{T}\right)^2.$$

Finally, the Mean Square Error (MSE) is a combination of the above two measures:

$$MSE = \frac{1}{R}\sum_{r=1}^{R} \left(T_r - \theta\right)^2.$$

An important relationship connecting these three measures is

$$MSE = bias^2 + variance.$$

Less commonly used criteria include the median bias and the median absolute deviation of T, where we use the median $\tilde{T}$ of our estimates rather than the mean $\bar{T}$.

All these criteria are listed in the table below.

Criterion	Definition	Estimate
Bias	$\mathrm{E}(T) - \theta$	$\bar{T} - \theta$
Median bias	$\mathrm{M}(T) - \theta$	$\tilde{T} - \theta$
Variance	$\mathrm{E}\left[(T - \mathrm{E}(T))^2\right]$	S_T^2
MSE	$\mathrm{E}\left[(T - \theta)^2\right]$	$\left(\bar{T} - \theta\right)^2 + S_T^2$
MAD	$\mathrm{M}\left[\lvert T - \theta \rvert\right]$	$\left[\lvert T - \theta \rvert\right]_{R/2}$

- Bias and median bias are measures of whether the estimator is systematically higher or lower than the target parameter.
- Variance is a measure of the **precision** of the estimator—that is, how far it deviates *from its average*. We might look at the square root of this, to assess the precision in the units of the original measure.
- Mean-squared error is a measure of **overall accuracy**, i.e. is a measure how far we typically are from the truth. We more frequently use the root mean-squared error, or RMSE, which is just the square root of the MSE.
- The median absolute deviation is another measure of overall accuracy that is less sensitive to an occasional bad mistake. In general the RMSE can be driven up by a single bad egg. The MAD is less sensitive to this.

For absolute assessments of performance, an estimator with low bias, low variance, and thus low RMSE is desired. For comparisons of relative performance, an estimator with lower RMSE is usually preferable to an estimator with higher RMSE; if two estimators have comparable RMSE, then the estimator with lower bias (or median bias) would usually be preferable.

It is important to recognize that the above performance measures depend on the scale of the parameter. For example, if our estimators are measuring a treatment impact in dollars, then our bias would be in dollars. Our variance and MSE would be in dollars squared, so we might take their square roots to put them back on the dollars scale.

Usually in a simulation, the scale of the outcome is irrelevant as we are comparing one estimator to the other. To ease interpretation, we might want to assess estimators relative to the baseline variation. To achieve this, we can generate data so the outcome has unit variance (i.e., we generate *standardized*

data). Then the bias, median bias, and root mean-squared error would all be in standard deviation units.

Furthermore, changing the scale of a parameter can lead to nonlinear changes in the performance measures. For instance, suppose that θ is a measure of the proportion of time that a behavior occurs. A natural way to transform this parameter would be to put it on the log-odds (logit) scale. However, because of the nonlinear aspect of the logit,

$$\text{Bias}\left[\text{logit}(T)\right] \neq \text{logit}\left(\text{Bias}[T]\right), \qquad \text{MSE}\left[\text{logit}(T)\right] \neq \text{logit}\left(\text{MSE}[T]\right),$$

and so on. This is fine, but one should be aware that this can happen and do it on purpose.

7.4 Assessing a standard error estimator

Statistics is perhaps more about assessing how good an estimate is than making an estimate in the first place. This translates to simulation studies: we generally not only want to know whether our estimator is doing a good job, but we often want to know whether we are able to get a good standard error for that estimator as well.

We first would compare the expected value of $\widehat{SE}$ to the actual SE. This tells us whether our uncertainty estimates are biased. We could also examine the standard deviation of $\widehat{SE}$, which tells us whether our estimates of uncertainty are relatively stable (especially compared to other methods). We finally could examine whether there is correlation between $\widehat{SE}$ and actual error (e.g., $|T - \theta|$). Good estimates of uncertainty should predict error in a given context (especially if calculating in conditional estimates). See **?**.

For the first assessment, we usually assess the quality of a standard error estimator with a relative performance criteria, rather than an absolute one as we saw above. For an example, suppose that in our simulation we are examining the performance of a point-estimator T for a parameter θ along with an estimator $\widehat{SE}$ for the standard error of T. In this case, we likely do not know the true standard error of T, for our simulation context, prior to the simulation. However, we can use the variance of T across the replications (S_T^2) to directly estimate the true sampling variance $\text{Var}(T) = SE^2(T)$. The *relative bias* of $\widehat{SE}^2$ would then be estimated by $RB = \bar{V}/S_T^2$, where $\bar{V}$ is the average of $\widehat{SE}^2$ across simulation runs. Note that a value of 1 for relative bias corresponds to exact unbiasedness. The relative bias measure is a measure of *proportionate* under- or over-estimation. For example, a relative bias of 1.12 would mean the standard error was, on average, 12% too large.

For parameters such as these, that measure scale, or that are always strictly positive, it often makes sense to quantify performance using such *relative* criteria. Relative criteria are very similar to the absolutre criteria discussed for

point estimators, but are defined as proportions of the target parameter, rather than as differences. Relative criteria are also often used when, for example, estimating quantities such as standard deviations. The table below defines several relative performance criteria.

Criterion	Definition	Estimate
Relative bias	$\mathrm{E}(T)/\theta$	$\bar{T}/\theta$
Relative median bias	$\mathrm{M}(T)/\theta$	$\tilde{T}/\theta$
Relative MSE	$\mathrm{E}\left[(T-\theta)^2\right]/\theta^2$	$\frac{(\bar{T}-\theta)^2+S_T^2}{\theta^2}$

7.4.1 Why not assess $widehatSE$ directly?

We typically see assessment of $\widehat{SE}^2$, not $\widehat{SE}$. In other words, we typically work with assessing whether the variance estimator is unbiased, etc., rather than the standard error estimator. This comes out of a few reasons. First, in practice, so-called unbiased standard errors usually are not in fact actually unbiased. For linear regression, for example, the classic standard error estimator is an unbiased *variance* estimator, meaning that we have a small amount of bias due to the square-rooting as:

$$E[\sqrt{V}] \neq \sqrt{E[V]}.$$

Variance is also the component that gives us the classic bias-variance breakdown of $ MSE = Variance + Bias^2$, so if we are trying to assign whether an overall MSE is due to instability or systematic bias, operating in this squared space may be preferable.

That being said, to put things in terms of performance criteria humans understand it is usually nicer to put final evaluation metrics back into standard error units. For example, saying there is a 10% reduction in the standard error is more meaningful (even if less impressive sounding) than saying there is a 19% reduction in the variance.

7.5 Assessing a hypothesis testing procedure

When hypothesis tests are used in practice, the researcher specifies a null (e.g., no treatment effect), collects data, and generates a p-value which is a measure of how extreme the observed data are from what we would expect to naturally occur, if the null were true. When we assess a method for hypothesis testing, we are therefore typically concerned with two aspects: *validity* and *power*.

7.5.1 Validity

Validity revolves around whether we erroneously reject the null when it is in fact true more than we should. Put another way, we say an inference method is valid if it has no more than an α chance of rejecting the null when we are testing at the α level. This means if we used this method 1000 times, where the null was true for all of those 1000 times, we should not see more than about 1000α rejections (so, 50, if we were using the classic $\alpha = 0.05$ rule).

To do this we would specify a data generating process where the null is in fact true. We then, for a series of such data sets with a true null, conduct our inferential processes on the data, record the p-value, and score whether we reject the null hypothesis or not.

We might then test our methods by exploring more extreme data generation processes, where the null is true but other aspects of the data (such as outliers or heavy skew) make estimation difficult. This allows us to understand if our methods are robust to strange data patterns in finite sample contexts.

The key concept for validity is that the date we generate, no matter how we do it, is data with a true null. We then check to see if we reject the null more than we should.

7.5.2 Power

Power is, loosely speaking, how often we notice an effect when one is there. This is a much more nebulous concept, because some effects are clearly easier to notice than others. Regardless of the estimator used, if the effect is large enough, we would notice. If we are comparing estimators to each other, this is less of a concern, because we are typically interested an relative performance. That being said, in order to generate data for a power evaluation, we have to generate data where there is something to detect. In other words, we need to commit to what the alternative is, and this can be a tricky business.

Typically, it is best to think of power as a function of sample size or effect size. Therefore, we will typically examine a sequence of scenarios with steadily increasing sample size or effect size, estimating the power for each scenario in the sequence. We then, for each sample in our series, estimate the power by the same process as for Validity, above. For each series we can then plot power curves, as we saw with some of our earlier vignettes.

When assessing validity, we want rejection rates to be low, below α, and when assessing power we want them to be as high as possible. But the simulation process itself, other than the data generating process, is exactly the same.

To put some technical terms to this framing, for both validity and power assessment the main performance criterion is the **rejection rate** of the hypothesis test. When the data are simulated from a model in which the null hypothesis

being tested is true, then the rejection rate is equivalent to the **Type-I error rate** of the test. When the data are simulated from a model in which the null hypothesis is false, then the rejection rate is equivalent to the **power** of the test (for given, non-null parameter values). Ideally, a testing procedure should have actual Type-I error equal to the nominal level α (this is the definition of validity), but such exact tests are rare.

There are some different perspectives on how close the actual Type-I error rate should be in order to qualify as suitable for use in practice. Following a strict statistical definition, a hypothesis testing procedure is said to be **level-α** if its actual Type-I error rate is *always* less than or equal to α. Among a set of level-α tests, the test with highest power would be preferred. If looking only at null rejection rates, then the test with Type-I error closest to α would usually be preferred. A less stringent criteria is sometimes used instead, where type I error would be considered acceptable if it is within 50% of the desired α.

Often, it is of interest to evaluate the performance of the test at several different α levels. A convenient way to calculate a set of different rejection rates is to record the simulated p-values and then calculate from those. To illustrate, suppose that P_r is the p-value from simulation replication k, for $k = 1, ..., R$. Then the rejection rate for a level-α test is defined as $\rho_\alpha = \Pr(P_r < \alpha)$ and estimated as

$$r_\alpha = \frac{1}{R}\sum_{r=1}^{R} I(P_r < \alpha).$$

7.6 Assessing confidence intervals

Some estimation procedures result in confidence intervals (or sets) which are ranges of values that should contain the true answer with some specified degree of confidence. For example, a normal-based confidence interval is a combination of an estimator and it's estimated uncertainty.

We typically score a confidence interval along two dimensions, **coverage rate** and the **average length**. To calculate coverage rate, we score whether each interval "captured" the true parameter. A success is if the true parameter is inside the interval. To calculate average length, we record each confidence interval's length, and then average across simulation runs. We say an estimator has good properties if it has good coverage, i.e. it is capturing the true value at least $1 - \alpha$ of the time, and if it is generally short (i.e., the average length of the interval is less than the average length for other methods).

Note that confidence interval coverage is simultaneously evaluating the estimators in terms of how well they estimate (precision) and their inferential properties. We have combined inference and estimation here.

Suppose that the confidence intervals are for the target parameter θ and have coverage level β. Let A_r and B_r denote the lower and upper end-points of the

confidence interval from simulation replication k, and let $W_r = B_r - A_r$, all for $k = 1, ..., R$. The coverage rate and average length criteria are then as defined in the table below.

Criterion	Definition	Estimate
Coverage	$\omega_\beta = \Pr(A \leq \theta \leq B)$	$\frac{1}{R}\sum_{r=1}^{R} I(A_r \leq \theta \leq B_r)$
Expected length	$\mathrm{E}(W) = \mathrm{E}(B - A)$	$\bar{W} = \bar{B} - \bar{A}$

Just as with hypothesis testing, a strict statistical interpretation would deem a hypothesis testing procedure acceptable if it has actual coverage rate greater than or equal to β. If multiple tests satisfy this criterion, then the test with the lowest expected length would be preferable. Some analysts prefer to look at lower and upper coverage separately, where lower coverage is $\Pr(A \leq \theta)$ and upper coverage is $\Pr(\theta \leq B)$.

7.7 Uncertainty in our performance estimates (the MCSE)

Our performance criteria are defined as average performance across an infinite number of trials. Of course, in our simulations we only run a finite number, and estimate the performance criteria with the sample of trials we generate. For example, if we are assessing coverage across 100 trials, we calculate what fraction rejected the null for that 100. But due to random chance, we might see a higher, or lower proportion rejected just due to random chance than what we would see if we ran the simulation forever.

To account for this estimation uncertainty we would want to calculate associated uncertainty estimates to go with our point estimates of performance. We want to, in other words, treat our simulation results as a dataset in its own right.

In this section we discuss how to calculate standard errors for the various performance critera given above. We call these Monte Carlo Simulation Errors, or MCSEs. For many performance critera, calculating a MCSE is quite straightforward: we have a nice, independent and identically distributed set of measurements, and the statastical inference is straightforward. For some other performance criteria we have to be a bit more clever.

First, we list MCSE expressions for many of our straightforward performance measures on the table below. In reading the table, recall that, for an estimator T, we have S_T^2 being the variance of T across our simulation runs. We also have

- Sample skewness (standardized): $g_T = \frac{1}{RS_T^3}\sum_{r=1}^{R}\left(T_r - \bar{T}\right)^3$ - Sample kurtosis (standardized): $k_T = \frac{1}{RS_T^4}\sum_{r=1}^{R}\left(T_r - \bar{T}\right)^4$

Criterion	MCSE
Bias	$\sqrt{S_T^2/R}$
Median bias	-
Variance	$S_T^2\sqrt{\frac{k_T-1}{R}}$
MSE	$\sqrt{\frac{1}{R}\left[S_T^4(k_T-1)+4S_T^3 g_T\left(\bar{T}-\theta\right)+4S_T^2\left(\bar{T}-\theta\right)^2\right]}$
MAD	-
Relative bias	$\sqrt{S_T^2/\left(R\theta^2\right)}$
Relative median bias	-
Relative MSE	$\sqrt{\frac{1}{R\theta^2}\left[S_T^4(k_T-1)+4S_T^3 g_T\left(\bar{T}-\theta\right)+4S_T^2\left(\bar{T}-\theta\right)^2\right]}$
Power & Validity	$\sqrt{r_\alpha\left(1-r_\alpha\right)/R}$
Coverage	$\sqrt{\omega_\beta\left(1-\omega_\beta\right)/R}$
Expected length	$\sqrt{S_W^2/R}$

7.7.1 MCSE for variance estimators

Estimating the MCSE of the relative bias or relative MSE of a (squared) standard error estimator is complicated by the appearance of a sample quantity, S_T^2, in the denominator of the ratio. This renders the formula above unusable, technically speaking.

To properly assess the overall MCSE, we need to take the uncertainty of our denominator into account. One way to do so is to use the *jackknife* technique. Let $\bar{V}_{(j)}$ and $S^2_{T(j)}$ be the average squared standard error estimate and the true variance estimate calculated from the set of replicates ***that excludes replicate*** j, for $j = 1, ..., R$. The relative bias estimate, excluding replicate j would then be $\bar{V}_{(j)}/S^2_{T(j)}$. Calculating all R versions of this relative bias estimate and taking the variance yields the jackknife variance estimator:

$$MCSE\left(\widehat{SE}^2\right) = \frac{1}{R}\sum_{j=1}^{R}\left(\frac{\bar{V}_{(j)}}{S^2_{T(j)}} - \frac{\bar{V}}{S_T^2}\right)^2.$$

This would be quite time-consuming to compute if we did it by brute force. However, with a few algebra tricks we can find a much quicker way. The tricks come from observing that

$$\bar{V}_{(j)} = \frac{1}{R-1}\left(R\bar{V} - V_j\right)$$
$$S^2_{T(j)} = \frac{1}{R-2}\left[(R-1)S^2_T - \frac{R}{R-1}\left(T_j - \bar{T}\right)^2\right]$$

These formulas can be used to avoid re-computing the mean and sample variance from every subsample. Instead, you calculate the overall mean and overall variance, and then do a small adjustment with each jackknife iteration. You can even implement this with vector processing in R!

7.7.2 The simhelpers package and MCSEs

The `simhelper` package is designed to calculate MCSEs (and the performance metrics themselves) for you. It is easy to use: here is an example on the Welch dataset that the package provides:

```
library( simhelpers )
welch <- welch_res %>%
  filter( method == "t-test" ) %>%
  dplyr::select( -method, -seed, -iterations )
welch
```

```
## # A tibble: 8,000 x 8
##       n1    n2 mean_diff      est    var p_val lower_bound upper_bound
##    <dbl> <dbl>     <dbl>    <dbl>  <dbl> <dbl>       <dbl>       <dbl>
##  1    50    50         0  0.0258  0.0954 0.934      -0.587       0.639
##  2    50    50         0  0.00516 0.0848 0.986      -0.573       0.583
##  3    50    50         0 -0.0798  0.0818 0.781      -0.647       0.488
##  4    50    50         0 -0.0589  0.102  0.854      -0.692       0.574
##  5    50    50         0  0.0251  0.118  0.942      -0.658       0.708
##  6    50    50         0 -0.115   0.106  0.725      -0.761       0.531
##  7    50    50         0  0.157   0.115  0.645      -0.517       0.831
##  8    50    50         0 -0.213   0.121  0.543      -0.903       0.478
##  9    50    50         0  0.509   0.117  0.139      -0.169       1.19
## 10    50    50         0 -0.354   0.0774 0.206      -0.906       0.198
## # ... with 7,990 more rows
```

7.8 Absolute vs. relative performance measures

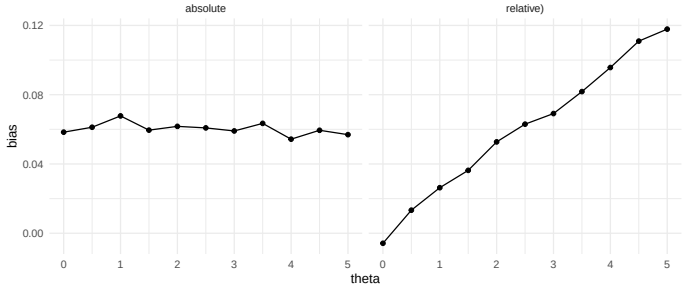
Depending on the model and estimation procedures being examined, a range of different criteria might be used to assess estimator performance. In particu-

lar, one might be deciding between using an absolute vs. relative performance measure. How does one pick?

In the above, we presented the absolute criteria for the point estimators and relative criteria for standard error estimators. But it turns out that this is not a fixed rule.

In general, we do not expect, for the performance (bias, variance, and MSE) of a point estimate such as a mean estimate to depend on its magnitude. In other words, if we are estimating some mean θ, and we generate data where $\theta = 100$ vs $\theta = 1000$ (or any arbitrary number), we would not generally expect that to change the magnitude of its bias, variance, or MSE. On the other hand, these different θs will have a large impact on the *relative* bias and *relative* MSE. (Want smaller relative bias? Just add a million to the parameter!) For these sorts of "location parameters" we generally use absolute measures of performance.

That being said, a more principled approach for determining whether to use absolute or relative performance criteria depends on assessing performance for *multiple* values of the parameter. In many simulation studies, replications are generated and performance criteria are calculated for several different values of a parameter, say $\theta = \theta_1, ..., \theta_p$. Let's focus on bias for now, and say that we've estimated (from a large number of replications) the bias at each parameter value.



If the absolute bias is roughly the same for all values of θ (as in the plot on the left), then it makes sense to report absolute bias as the summary performance criterion. On the other hand, if the bias grows roughly in proportion to θ (as in the plot on the right), then relative bias is a better summary criterion.

More broadly, one can calculate performance relative to some baseline. For example, if one of the estimators is the "generic method", we could calculate ratios of the RMSE of our estimators to the baseline RMSE. This can provide a way of standardizing across simulation scenarios where the overall scale of the RMSE changes radically. While a powerful tool, it is not without risks: if you scale relative to something, then higher or lower ratios can either be due to the primary method of interest (the numerator) or behavior of the reference method in the denominator. These relative ratios can end up being confusing to interpret due to this tension.

7.9 Exercises: Simulating Cronbach's alpha

Cronbach's α coefficient is commonly reported as a measure of the internal consistency among a set of test items. Consider a set of p test items with population variance-covariance matrix $\Phi = \left[\phi_{ij}\right]_{i,j=1}^{p}$. This population variance-covariance matrix describes how our p test items co-vary.

Cronback's α is, under this model, defined as

$$\alpha = \frac{p}{p-1}\left(1 - \frac{\sum_{i=1}^{p}\phi_{ii}}{\sum_{i=1}^{p}\sum_{j=1}^{p}\phi_{ij}}\right).$$

Given a sample of size n, the usual estimate of α is obtained by replacing the population variances and covariances with corresponding sample estimates. Letting s_{ij} denote the sample covariance of items i and j

$$A = \frac{p}{p-1}\left(1 - \frac{\sum_{i=1}^{p}s_{ii}}{\sum_{i=1}^{p}\sum_{j=1}^{p}s_{ij}}\right).$$

If we assume that the items follow a multivariate normal distribution, then A corresponds to the maximum likelihood estimator of α.

In these exercises, we will examine the properties of this estimator when the set of P items is *not* multi-variate normal, but rather follows a multivariate t distribution with v degrees of freedom. For simplicity, we shall assume that the items have common variance and have a **compound symmetric** covariance matrix, such that $\phi_{11} = \phi_{22} = \cdots = \phi_{pp} = \phi$ and $\phi_{ij} = \rho\phi$. In this case we can simplify our expression for α to

$$\alpha = \frac{p\rho}{1 + \rho(p-1)}.$$

7.9.1 The data-generating function

The following function generates a sample of n observations of p items from a multivariate t distribution with a compound symmetric covariance matrix, intra-class correlation ρ, and v degrees of freedom:

```
library(mvtnorm)

r_mvt_items <- function(n, p, icc, df) {
  V_mat <- icc + diag(1 - icc, nrow = p)
  X <- rmvt(n = n, sigma = V_mat, df = df)
  colnames(X) <- LETTERS[1:p]
```

```
  X
}

small_sample <- r_mvt_items(n = 8, p = 3, icc = 0.7, df = 5)
small_sample
```

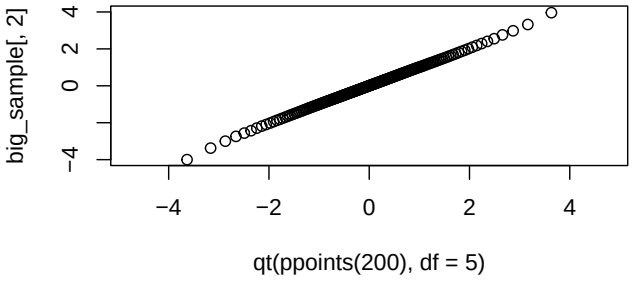
```
##                A          B           C
## [1,]  0.09382233 -0.3954138 -0.07444013
## [2,] -0.26192949 -0.4419227 -0.39160382
## [3,]  0.39158338  0.9757591 -0.53170400
## [4,] -0.05918679 -1.5391954 -0.79945929
## [5,] -1.84261034  0.1107519 -1.08034605
## [6,]  0.39922138  0.3197069  0.59206764
## [7,] -0.19669418 -0.6172658 -1.43238613
## [8,] -0.75393141 -0.8665165  0.03773137
```

To check that the function is indeed simulating data following the intended distribution, let's generate a very large sample of items. We can then verify that the correlation matrix of the items is compound-symmetric and that the marginal distributions of the items follow t distributions with specified degrees of freedom.

```
big_sample <- r_mvt_items(n = 100000, p = 4, icc = 0.7, df = 5)

round(cor(big_sample), 3) # looks good
```

```
##       A     B     C     D
## A 1.000 0.701 0.702 0.702
## B 0.701 1.000 0.700 0.703
## C 0.702 0.700 1.000 0.700
## D 0.702 0.703 0.700 1.000
```

```
qqplot(qt(ppoints(200), df = 5), big_sample[,2], ylim = c(-4,4))
```



7.9.2 The estimation function

van Zyl, Neudecker, and Nel (2000) demonstrate that, if the items have a compound-symmetric covariance matrix, then the asymptotic variance of A is

$$\text{Var}(A) \approx \frac{2p(1-\alpha)^2}{(p-1)n}.$$

Substituting A in place of α gives an estimate of the variance of A. The following function calculates A and its variance estimator from a sample of data:

```
estimate_alpha <- function(dat) {
  V <- cov(dat)
  p <- ncol(dat)
  n <- nrow(dat)

  # Calculate A with our formula
  A <- p / (p - 1) * (1 - sum(diag(V)) / sum(V))

  # Calculate our estimate of the variance (SE^2) of A
  Var_A <- 2 * p * (1 - A)^2 / ((p - 1) * n)

  # Pack up our results
  data.frame(A = A, Var = Var_A)
}

estimate_alpha(small_sample)
```

```
##           A        Var
## 1 0.4922233 0.09668892
```

The `psych` package provides a function for calculating α, which can be used to verify that the calculation of A in `estimate_alpha` is correct:

```
library(psych)
```

```
##
## Attaching package: 'psych'
```

```
## The following objects are masked from 'package:ggplot2':
##
##     %+%, alpha
```

```
summary(alpha(x = small_sample))$raw_alpha
```

```
## Number of categories should be increased  in order to count frequencies.
```

```
##
## Reliability analysis
##  raw_alpha std.alpha G6(smc) average_r S/N  ase  mean   sd median_r
##       0.49       0.5    0.43      0.25   1 0.32 -0.35 0.51      0.2
```

```
## NULL
```

7.9.3 Replicates

The following function generates a specified number of replicates of A and its variance estimator, for user-specified parameter values n, p, α, and v:

```
library(dplyr)

simulate_alpha <- function(reps, n, p, alpha, df) {
  icc <- alpha / (p - alpha * (p - 1))
  replicate(reps, {
    dat <- r_mvt_items(n = n, p = p, icc = icc, df = df)
    estimate_alpha(dat)
  }, simplify = FALSE) %>%
  bind_rows()
}
```

We can use it to generate 1000 replicates using samples of size $n = 40$, $p = 6$ items, a true $\alpha = 0.8$, and $v = 5$ degrees of freedom:

```
reps <- 1000
alpha_true <- 0.8
alpha_reps <- simulate_alpha(reps = reps, n = 40, p = 6, alpha = alpha_true, df = 5)
head(alpha_reps)
```

```
##           A         Var
## 1 0.8030085 0.002328340
## 2 0.8023319 0.002344362
## 3 0.8328883 0.001675579
## 4 0.7354949 0.004197777
## 5 0.6870548 0.005876084
## 6 0.7876845 0.002704671
```

7.9.4 Estimator performance

1. With the parameters specified above, calculate the bias of A. Also calculate the Monte Carlo standard error (MCSE) of the bias estimate.
2. Calculate the mean squared error of A, along with its MCSE.
3. Calculate the relative bias of the asymptotic variance estimator.
4. **(Challenge problem)** Code up a jackknife MCSE function to calculate the MCSE for the relative bias of the asymptotic variance estimator. Let it use the following template that takes a vector of point estimates and associated standard errors.

```
jackknife_MCSE <- function( T, SE ) {
  # code
}
```

5. **(Challenge problem)** Make a `run_simulation()` function that uses `simulate_alpha` and returns a one-row data frame with columns corresponding to the bias, mean squared error, and relative bias of the asymptotic variance estimator. Use the function to evaluate the performance of A for true values of α ranging from 0.5 to 0.9 (i.e., `alpha_true_seq <- seq(0.5, 0.9, 0.1)`).

7.9.5 Confidence interval coverage

One way to obtain an approximate confidence interval for α would be to take $A \pm z\sqrt{\text{Var}(A)}$, where $\text{Var}(A)$ is estimated as described above and z is a standard normal critical value at the appropriate level (i.e., $z = 1.96$ for a 95% CI).

However, van Zyl, Neudecker, and Nel (2000) suggest that a better approximation involves first applying a transformation to A (to make it more normal in shape), then calculating a confidence interval, then back-transforming to the original scale (this is very similar to the procedure for calculating confidence intervals for correlation coefficients, using Fisher's z transformation). Let

$$\beta = \frac{1}{2}\ln(1-\alpha)$$
$$B = \frac{1}{2}\ln(1-A)$$

and

$$V^B = \frac{p}{2n(p-1)}.$$

An approximate confidence interval for β is given by $[B_L, B_U]$, where

$$B_L = B - z\sqrt{V^B}, \qquad B_U = B + z\sqrt{V^B}.$$

Applying the inverse of the transformation gives a confidence interval for α:

$$\left[1 - \exp(2B_U),\ 1 - \exp(2B_L)\right].$$

6. Modify the `estimate_alpha` function to calculate a confidence interval for α, following the method described above.

7. With the modified version of `estimate_alpha`, re-run the following code to obtain 1000 replicated confidence intervals. Calculate the true coverage rate of the confidence interval. Also calculate the Monte Carlo standard error (MCSE) of this coverage rate.

```
reps <- 1000
alpha_true <- 0.8
alpha_reps <- simulate_alpha(reps = reps, n = 40, p = 6, alpha = alpha_true, df = 5)
```

8. Calculate the average length of the confidence interval for α, along with its MCSE.

9. Compare the results of this approach to a more naive approach. Are there gains in performance?

Chapter 8

Case study: Cronbach Alpha

In this section we walk through the case study of Cronbach Alpha to illustrate the filling out of the code skeleton we get from `simhelpers`'s `create_skeleton()` package.

We first create the skeleton, and then start filling in the pieces.

```
library( simhelpers )
create_skeleton()
```

8.1 Data-generating model

The first two sections in the skeleton are about the data-generating model:

```
rm(list = ls())

#--------------------------------------------------------
# Set development values for simulation parameters
#--------------------------------------------------------

# What are your model parameters?
# What are your design parameters?

#--------------------------------------------------------
# Data Generating Model
#--------------------------------------------------------
```

```
dgm <- function(model_params) {

  return(dat)
}

# Test the data-generating model - How can you verify that it is correct?
```

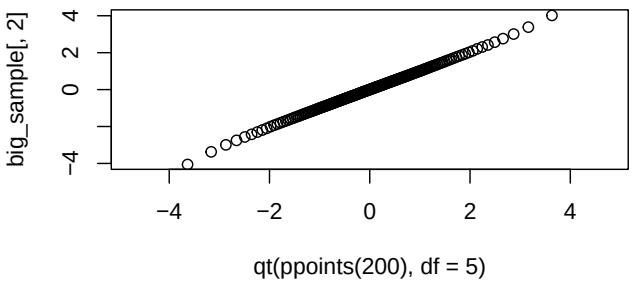
Here, we need to create and test a function that takes model parameters (and sample sizes and such) as inputs, and produces a simulated dataset. For the Cronbach alpha simulation, the function looks like this:

```
library(mvtnorm)
rm(list = ls())

#---------------------------------------------------------
# Set development values for simulation parameters
#---------------------------------------------------------

# model parameters
alpha <- 0.73 # true alpha
df <- 12 # degrees of freedom

# design parameters
n <- 50 # sample size
p <- 6 # number of items

#---------------------------------------------------------
# Data Generating Model
#---------------------------------------------------------

r_mvt_items <- function(n, p, alpha, df) {
  icc <- alpha / (p - alpha * (p - 1))
  V_mat <- icc + diag(1 - icc, nrow = p)
  X <- rmvt(n = n, sigma = V_mat, df = df)
  colnames(X) <- LETTERS[1:p]
  X
}

# Test the data-generating model

big_sample <- r_mvt_items(n = 100000, p = 4, alpha = 0.73, df = 5)
round(cor(big_sample), 3) # looks good
```

```
##       A     B     C     D
```

```
## A 1.000 0.406 0.411 0.406
## B 0.406 1.000 0.403 0.400
## C 0.411 0.403 1.000 0.405
## D 0.406 0.400 0.405 1.000
```

```
qqplot(qt(ppoints(200), df = 5), big_sample[,2], ylim = c(-4,4))
```



8.2 Estimation procedures

The next section of the template looks like this:

```
#------------------------------------------------------
# Model-fitting/estimation/testing functions
#------------------------------------------------------

estimate <- function(dat, design_params) {

  return(result)
}

# Test the estimation function
```

Here, we need to create a function that takes simulated data as input (and possibly also design parameters, like sample size), and produces a set of estimates (or confidence intervals, or p-values, etc.). As usual, we should also test that the function is correct.

Here's what this code looks like for the Cronbach alpha simulation:

```
#------------------------------------------------------
# Model-fitting/estimation/testing functions
#------------------------------------------------------

estimate_alpha <- function(dat, coverage = .95) {
  V <- cov(dat)
  p <- ncol(dat)
  n <- nrow(dat)
  A <- p / (p - 1) * (1 - sum(diag(V)) / sum(V))
  Var_A <- 2 * p * (1 - A)^2 / ((p - 1) * n)

  B <- log(1 - A) / 2
  SE_B <- sqrt(p / (2 * n * (p - 1)))
  z <- qnorm((1 - coverage) / 2)
  CI_B <- B + c(-1, 1) * SE_B * z
  CI_A <- 1 - exp(2 * CI_B)

  data.frame(A = A, Var_A = Var_A, CI_L = CI_A[1], CI_U = CI_A[2])
}

# Test the estimation function
small_sample <- r_mvt_items(n = 50, p = 6, alpha = 0.73, df = 5)
estimate_alpha(small_sample)
```

```
##           A       Var_A      CI_L      CI_U
## 1 0.6776348 0.004988126 0.5047357 0.7901741
```

The function takes a simulated dataset as input and spits out a point estimate of alpha, an estimate of the variance of alpha, and a confidence interval for alpha (at the 95% coverage level, by default).

We've already seen how to use the `replicate` function to generate a whole bunch of simulated estimates:

```
alpha_sims <-
  replicate(n = 10, {
    dat <- r_mvt_items(n = 50, p = 6, alpha = 0.73, df = 5)
    estimate_alpha(dat)
  }, simplify = FALSE) %>%
  bind_rows()

alpha_sims
```

```
##           A        Var_A       CI_L      CI_U
## 1  0.7376881 0.0033027608  0.5969983 0.8292625
```

```
## 2  0.5774827 0.0085690016  0.3508674 0.7249855
## 3  0.6474001 0.0059676806  0.4582847 0.7704944
## 4  0.2444404 0.0274017727 -0.1608006 0.5082099
## 5  0.7237283 0.0036636500  0.5755512 0.8201761
## 6  0.7341302 0.0033929646  0.5915321 0.8269466
## 7  0.7961477 0.0019946770  0.6868124 0.8673135
## 8  0.5563208 0.0094488599  0.3183554 0.7112113
## 9  0.8823696 0.0006641713  0.8192791 0.9234350
## 10 0.6132132 0.0071809934  0.4057618 0.7482423
```

8.3 Performance calculations

The next section of the template deals with performance calculations:

```
#------------------------------------------------------
# Calculate performance measures
# (For some simulations, it may make more sense
# to do this as part of the simulation driver.)
#------------------------------------------------------

performance <- function(results, model_params) {

  return(performance_measures)
}

# Check performance calculations
```

The `performance()` function takes as input a bunch of simulated data (which we might call `results`) and the true values of the model parameters (`model_params`) and returns as output a set of summary performance measures. As noted in the comments above, for simple simulations it might not be necessary to write a separate function to do these calculations. For more complex simulations, though, it can be helpful to break these calculations out in a function.

For the Cronbach alpha simulation, we might want to calculate the following performance measures:

- bias and root mean-squared error (RMSE) of the alpha point estimate
- relative bias of the variance estimator
- coverage of the confidence interval

Here is a function that calculates these measures (along with Monte Carlo standard errors), given a data frame containing model results. It uses the jackknife technique to get Monte Carlo standard errors for the RMSE and relative bias.

```r
#---------------------------------------------------
# Calculate performance measures
#---------------------------------------------------

alpha_performance <- function(alpha_sims, alpha, coverage_level = .95) {

  # setup
  K <- nrow(alpha_sims)
  A_err <- alpha_sims$A - alpha
  var_A <- var(alpha_sims$A)

  # bias
  A_bias <- mean(A_err)
  A_bias_MCSE <- sqrt(var_A / K)

  # RMSE
  A_RMSE <- sqrt(mean((A_err)^2))
  RMSE_j <- sqrt((A_RMSE^2 * K - A_err^2) / (K - 1))
  A_RMSE_MCSE <- sd(RMSE_j)

  # relative bias of variance estimator
  V_bar <- mean(alpha_sims$Var_A)
  V_j <- (V_bar * K - alpha_sims$Var_A) / (K - 1)
  Ssq_j <- ((K - 1) * var_A - A_err^2 * K / (K - 1)) / (K - 2)
  RB_j <- V_j / Ssq_j
  V_relbias <- V_bar / var_A
  V_relbias_MCSE <- sd(RB_j)

  # coverage
  coverage <- mean(alpha_sims$CI_L < alpha & alpha < alpha_sims$CI_U)
  coverage_MCSE <- sqrt(coverage_level * (1 - coverage_level) / K)

  data.frame(
    criterion = c("alpha bias","alpha RMSE", "V relative bias", "coverage"),
    est = c(A_bias, A_RMSE, V_relbias, coverage),
    MCSE = c(A_bias_MCSE, A_RMSE_MCSE, V_relbias_MCSE, coverage_MCSE)
  )
}

# Check performance calculations
alpha_performance(alpha_sims, alpha = 0.73)
```

```
##         criterion         est       MCSE
## 1      alpha bias -0.07870788 0.05531869
## 2      alpha RMSE  0.18367458 0.02678076
```

```
## 3 V relative bias  0.23393085 0.85212103
## 4        coverage  0.60000000 0.06892024
```

8.4 Simulation driver

We now have all the components we need to get simulation results, given a set of parameter values. In the next section of the template, we put all these pieces together in a function—which we might call the *simulation driver*—that takes as input 1) parameter values, 2) the desired number of replications, and 3) optionally, a seed value. The function produces as output a single set of performance estimates. Generically, the function looks like this:

```
#------------------------------------------------------------
# Simulation Driver - should return a data.frame or tibble
#------------------------------------------------------------

runSim <- function(iterations, model_params, design_params, seed = NULL) {
  if (!is.null(seed)) set.seed(seed)

  results <- replicate(iterations, {
                dat <- dgm(model_params)
                estimate(dat, design_params)
              })

  performance(results, model_params)
}

# demonstrate the simulation driver
```

The `runSim` function should require very little modification for a new simulation. Essentially, all we need to change is the names of the functions that are called, so that they line up with the functions we have designed for our simulation. Here's what this looks like for the Cronbach alpha simulation:

```
#------------------------------------------------------------
# Simulation Driver - should return a data.frame or tibble
#------------------------------------------------------------

run_alpha_sim <- function(iterations, n, p, alpha, df, coverage = 0.95, seed = NULL) {

  if (!is.null(seed)) set.seed(seed)

  results <-
    replicate(n = iterations, {
```

```
      dat <- r_mvt_items(n = n, p = p, alpha = alpha, df = df)
      estimate_alpha(dat, coverage = coverage)
    }, simplify = FALSE) %>%
    bind_rows()

  alpha_performance(results, alpha = alpha, coverage = coverage)
}
```

Now to run our simulation, we just call our simulation driver.

8.5 Running the simulation

In the previous sections, we've created code that will generate a set of performance estimates, given a set of parameter values. We've also created a dataset that represents every combination of parameter values that we want to examine. How do we put the pieces together?

If we only had a couple of parameter combinations, it would be easy enough to just call our `run_alpha_sim` function a couple of times:

```
run_alpha_sim(iterations = 100, n = 50, p = 4, alpha = 0.7, df = 5)
```

```
##         criterion          est         MCSE
## 1      alpha bias -0.001593413 0.0088752547
## 2      alpha RMSE  0.088322044 0.0006681178
## 3 V relative bias  0.668658274 0.0089503252
## 4        coverage  0.850000000 0.0217944947
```

```
run_alpha_sim(iterations = 100, n = 100, p = 4, alpha = 0.7, df = 5)
```

```
##         criterion         est         MCSE
## 1      alpha bias -0.01800913 0.0079419752
## 2      alpha RMSE  0.08104783 0.0009785838
## 3 V relative bias  0.45395388 0.0111150091
## 4        coverage  0.84000000 0.0217944947
```

```
run_alpha_sim(iterations = 100, n = 50, p = 8, alpha = 0.7, df = 5)
```

```
##         criterion         est         MCSE
## 1      alpha bias -0.01317160 0.0084111506
## 2      alpha RMSE  0.08472006 0.0006497369
## 3 V relative bias  0.67898948 0.0092084481
## 4        coverage  0.89000000 0.0217944947
```

```
run_alpha_sim(iterations = 100, n = 100, p = 8, alpha = 0.7, df = 5)
```

```
##          criterion          est         MCSE
## 1      alpha bias -0.008556854 0.0065019050
## 2      alpha RMSE  0.065256585 0.0005078102
## 3 V relative bias  0.537396045 0.0078483658
## 4        coverage  0.840000000 0.0217944947
```

But in an actual simulation we will probably have too many different combinations to do this "by hand." The final sections of the simulation template demonstrate two different approaches to doing the calculations for *every* combination of parameter values, given a set of parameter values one wants to explore.

This is discussed further in Chapter @ref(exp_design).

Chapter 9

Ensuring reproducibility

In the prior section we built a simulation driver. Because this function involves generating random numbers, re-running it with the exact same input parameters will still produce different results:

```
run_alpha_sim(iterations = 10, n = 50, p = 6, alpha = 0.73, df = 5)
```

```
##         criterion         est       MCSE
## 1      alpha bias 0.009661492 0.03483168
## 2      alpha RMSE 0.104940732 0.01287825
## 3 V relative bias 0.311344712 0.11332344
## 4        coverage 0.800000000 0.06892024
```

```
run_alpha_sim(iterations = 10, n = 50, p = 6, alpha = 0.73, df = 5)
```

```
##         criterion          est        MCSE
## 1      alpha bias -0.004851553 0.023362490
## 2      alpha RMSE  0.070255184 0.003544186
## 3 V relative bias  0.707553736 0.101756695
## 4        coverage  0.800000000 0.068920244
```

Of course, using a larger number of iterations will give us more precise estimates of the performance criteria. If we want to get the *exact* same results, however, we have to control the random process.

This is more possible than it sounds: Monte Carlo simulations are random, but computers are not. When we generate "random numbers" they actually come from a chain of mathematical equations that, given a number, will generate the next number in a deterministic sequence. Given that number, it will generate

the next, and so on. The numbers we get back are a part of this chain of (very large) numbers that, ideally, cycles through an extremely long list of numbers in a haphazard and random looking fashion.

This is what the `seed` argument that we have glossed over before is all about. If we set the same seed, we get the same results:

```
run_alpha_sim(iterations = 10, n = 50, p = 6, alpha = 0.73, df = 5, seed = 6)
```

```
##         criterion         est       MCSE
## 1      alpha bias -0.02053560 0.02585963
## 2      alpha RMSE  0.08025083 0.01344827
## 3 V relative bias  0.64909209 1.43035647
## 4        coverage  0.90000000 0.06892024
```

```
run_alpha_sim(iterations = 10, n = 50, p = 6, alpha = 0.73, df = 5, seed = 6)
```

```
##         criterion         est       MCSE
## 1      alpha bias -0.02053560 0.02585963
## 2      alpha RMSE  0.08025083 0.01344827
## 3 V relative bias  0.64909209 1.43035647
## 4        coverage  0.90000000 0.06892024
```

This is useful because it ensure the full reproducibility of the results. In practice, it is a good idea to always set seed values for your simulations, so that you (or someone else!) can exactly reproduce the results. Let's look more at how this works.

9.1 Seeds and pseudo-random number generators

In R, we can start a sequence of **deterministic** but **random-seeming** numbers by setting a "seed". This means all the random numbers that come after it will be the same.

Compare this:

```
rchisq(3, df = 5)
```

```
## [1] 1.193539 5.338936 6.294274
```

```
rchisq(3, df = 5)
```

```
## [1] 0.7189536 6.3697718 8.2913988
```

To this:

```
set.seed(20210527)
rchisq(3, df = 5)
```

```
## [1] 1.643698 6.229613 3.249164
```

```
set.seed(20210527)
rchisq(3, df = 5)
```

```
## [1] 1.643698 6.229613 3.249164
```

By setting the seed the second time we reset our sequence of random numbers. Similarly, to ensure reproducibility in our simulation, we add an option to set the seed value of the random number generator.

This seed is low-level, meaning if we are generating numbers via `rnorm` or `rexp` or `rchisq`, it doesn't matter. Each time we ask for a random number from the low-level pseudo-random number generator, it gives us the next number back. These other functions than just transform the number to be of the correct distribution.

9.2 Including seed in our simulation driver

The easy way to ensure reproducability is to pass a seed as a parameter. If we leave it NULL, we ignore it and just continue generating random numbers from wherever we are in the system. If we specify a seed, however, we set it at the beginning of the scenario, and then all the numbers that follow will be in sequence. Our code will typically look like this:

```
run_alpha_sim <- function(iterations, n, p, alpha, df, coverage = 0.95, seed = NULL) {

  if (!is.null(seed)) set.seed(seed)

  results <-
    replicate(n = iterations, {
      dat <- r_mvt_items(n = n, p = p, alpha = alpha, df = df)
```

```
      estimate_alpha(dat, coverage = coverage)
    }, simplify = FALSE) %>%
    bind_rows()

  alpha_performance(results, alpha = alpha, coverage = coverage)
}
```

Using our seed, we get identical results, as we saw in the intro, above.

9.3 Reasons for setting the seed

Reproducibility allows us to easily check if we are running the same code that generate the results in some report. It also helps with debugging. For example, say we had an error that showed up one in a thousand, causing our simulation to crash sometimes.

If we set a seed, and see that it crashes, we can then go try and catch the error and repair our code, and then rerun the simulation. If it runs clean, we know we got the error. If we had not set the seed, we would not know if we were just getting lucky, and avoiding the error.

Chapter 10

More on functions

This chapter is not about simulation, but does have a few tips and tricks regarding coding that are worth attending to.

10.1 Default arguments for functions

To generate easy-to-use, but easy to configure code, use default arguments. For example,

```
my_function = function( a = 10, b = 20 ) {
     100 * a + b
}

my_function()
```

```
## [1] 1020
```

```
my_function( 5 )
```

```
## [1] 520
```

```
my_function( b = 5 )
```

```
## [1] 1005
```

```
my_function( b = 5, a = 1 )
```

```
## [1] 105
```

We can call it when we don't know what the arguments are, but then when we know more about the function, we can specify things of interest. Lots of R commands work exactly this way, and for good reason.

Especially for code to generate random datasets, default arguments can be a lifesaver as you can call the method before you know exactly what everything means.

For example, consider the `blkvar` package that has some code to generate blocked randomized datasets. We might locate a promising method, and type it in:

```
library( blkvar )
generate_blocked_data()
```

```
## Error in generate_blocked_data(): argument "n_k" is missing, with no default
```

That didn't work, but let's provide some block sizes and see what happens:

```
generate_blocked_data( c( 3, 2 ) )
```

```
##    B         Y0       Y1
## 1 B1 -0.6039556 4.100378
## 2 B1  0.7658181 6.294909
## 3 B1  0.6020900 5.482969
## 4 B2 -1.9584589 3.974284
## 5 B2  0.1514932 6.773053
```

Nice! We see that we have a block ID and the control and treatment potential outcomes. We also don't see a random assignment variable, so that tells us we probably need some other methods as well. But we can play with this as it stands right away.

Next we can see that there are many things we might tune:

```
args( generate_blocked_data )
```

```
## function (n_k, sigma_alpha = 1, sigma_beta = 0, beta = 5, sigma_0 = 1,
##     sigma_1 = 1, corr = 0.5, exact = FALSE)
## NULL
```

The documentation will tell us more, but if we just need some sample data, we can quickly assess our method before having to do much reading and understanding. Only once we have identified what we need do we really need to turn to the documentation itself.

Chapter 11

Case study: A simulation with clustered data

Generating data with complex structure can be intimidating, but if you set out a recipe for how the data is generated it is often not to bad to build that recipe up with code. We will illustrate how to tackle this kind of data with a case study of best practices for analyzing data from a cluster-randomized RCT of students nested in schools.

A lot of the current literature on multisite trials is exploring how variation in the size of impacts across sites can cause bad things can happen. What does it mean for this particular context?

There are various ways of analyzing such data:

- Individual Level Analysis
  - Multilevel modeling (MLM): Fit a multilevel model to account for dependencies within cluster.
  - Linear regression (LR): Fit a linear model and use cluster robust standard errors.
- Aggregation
  - Aggregation (Agg): Calculate average outcomes for each cluster and fit a linear model with heteroskedastic robust SEs

We might then ask, are any of these strategies biased? When and how much? Are any of these strategies more precise (have smaller SEs)? Are the standard errors for these different strategies valid? We might think aggregation should be worse since we are losing information, right? If so, how much is lost?

To make this investigation a bit more rich, we are also going to ask a final question that will influence our data generating process. We want to investigate what happens when the impact of a site depends on the site size. This is a common question that has gained some attention in the education world, where we might reasonably think sites of different sizes may respond to treatment differently. We want to know if our studied methods would end up giving us biased results, should there be such a relationship.

11.1 A design decision: What do we want to manipulate?

There are a lot of ways we might generate data. To figure out what kinds of controls we have on that process, we need to think about the goals of the simulation.

In our case, for example, we might think:

1) We figure if all the sites are the same size, we are probably safe. But if sites vary, then we could have issues with our estimators.
2) Also, if site size varies, but has nothing to do with impact, then we are probably good, at least for bias, but if it is associated then how we average our sites is going to matter.

Usually it is good practice to keep the simple option along with the complex one. We want to both check that something does not matter as well as verify it does.

Given this, we land on the following points:

- We need to consider both all-same-size sites and variable size sites.
- Our DGP probably should have some impact variation across sites.
- We should probably connect impact variation to site size to explore a more malicious context.
- For simplicity we will simply include a site size by treatment interaction term to get our heterogeneity.

11.2 A mathematical model for cluster-randomized data

The easiest way to write down a recipe for data generation is with a mathematical model. This is especially important for more complex DGPs, such as those for hierarchical data.

We know we want a collection of clusters with different sizes and different baseline mean outcomes. To keep things simple, we might want a common treatment shift within cluster: if we treat a cluster, everyone is raised by some specified amount.

We want to have some measure of site size. For starters, lets create a covariate which is the percent of the average site size that a site is:

$$S_j = \frac{n_j - \bar{n}}{\bar{n}}$$

Using this coveriate, we could end up with this model to describe our data:

$$\begin{aligned} Y_{ij} &= \beta_{0j} + \epsilon_{ij} \\ \epsilon_{ij} &\sim N(0, \sigma^2_\epsilon) \\ \beta_{0j} &= \gamma_0 + \gamma_1 Z_j + \gamma_2 Z_j S_j + u_j \\ u_j &\sim N(0, \sigma^2_u) \end{aligned}$$

Our parameters are the mean outcome of control unit (γ_0), the treatment Impact (γ_1), the amount of cross site variation (σ^2_u), and residual variation (σ^2_ϵ). Our γ_2 is our site-size by treatment interaction term: bigger sites will (assuming γ_2 is positive) have larger treatment impacts.

If you prefer the reduced form, it would be:

$$Y_{ij} = \gamma_0 + \gamma_1 Z_j + \gamma_2 Z_j S_j + u_j + \epsilon_{ij}$$

We might also include a main effect for S_j. This would make larger sites systematically different than smaller sites at baseline, rather than having it only be part of our treatment variation term. For simplicity we drop it here.

To generate data, we would also need several other quantities specified. First, we need to know the number of clusters (J), the sizes of the clusters (n_j, for $j = 1, \ldots, J$). We have to provide a recipe for generating these sizes. We might try

$$n_j \sim unif((1-\alpha)\bar{n}, (1+\alpha)\bar{n})$$

with a fixed α to control the amount of variation in cluster size. If $\bar{n} = 100$ and $\alpha = 0.25$ then we would, for example, have sites ranging from 75 to 125 in size.

Given how we are generating site size, look again at our treatment impact heterogeneity term:

$$\gamma_2 Z_j \left(\frac{n_j - \bar{n}}{\bar{n}} \right)$$

Note how we are standardizing by average site size to make our covariate not change in terms of its importance as a function of site size, but rather as a function of α. In particular, $\frac{n_j - \bar{n}}{\bar{n}}$ will range from $-\alpha$ to α, regardless of average site size. Carefully setting up a DGP so the "knobs" we use are standardized like this can make interpreting the simulation results much easier. Consider if we did not divide by $\bar{n}$: then larger sites would also have more severe heterogeniety in treatment impact; this could make interpreting the results very confusing.

We next need to define how we generate our treatment indicator, Z_j. We might specify some proportion p assigned to treatment, and set $Z_j = 1$ or $Z_j = 0$ using a simple random sampling approach on our J units.

11.3 Multilevel data generation is a recipe using a statistical model

Now let's translate our mathematical model to code. In the real world:

- We obtain data, we pick a model, we estimate parameters
- The data comes with covariates and outcomes
- It also comes with sample size, sizes of the clusters, etc.

In the simulation world, by comparison:

- We pick a model, we decide how much data, we generate covariates, we pick the parameters, and then we generate outcomes
- We need to decide how many clusters, how big the clusters are, etc.
- We have to specify how the covariates are made. This piece is very different from real-world analysis.

Step 1: Generate your sites

- Generate site-level covariates
- Generate sample size within each site
- Generate site level random effects

Step 2: Generate your students inside the sites

- Generate student covariates
- Generate student residuals
- Add everything up to generate student outcomes

The mathematical model gives us exactly the details we need to execute on these steps. In particular, we can translate the math directly to R code, and then finally put it all in a function.

In general, we have several components to our model:

COVARIATES and STRUCTURAL COVARIATES Covariates are the things that we are usually given when analyzing real data. This is a broad definition, including things beside baseline information:

- Conventional: student demographics, school-level characteristics, treatment assignment
- Structural: number of observations in each school, proportion treated in each school

We don't often think of these things as "covariates" but in a simulation we have to get them from somewhere.

MODEL This is the parametric relationship between everything: how the outcomes are linked to the covariates. This includes specification of any additional randomness (residuals, etc.)

DESIGN PARAMETERS These are, e.g., the number of sites or variation in site size. These control how we generate the structural covariates. In the real world, we don't tend to think of these things are covariates per se, they are more just consequences of the data. We rarely model them, but instead conidition on them, in a statistical analysis.

PARAMETERS These are the specifics: for a given model, parameters describe degree of variability, what the slope is, and so forth. We usually estimate these FROM data. Critically, if we know them, we can GENERATE NEW DATA.

11.3.1 Generating the multisite data

We know we will need to generate and then analyze data. First lets focus on the "generate" piece. Borrowing from our DGP skeleton, we specify a function with all the parameters we might want to pass it, including defaults for each (see @(#default_arguments) for more on function defaults):

```
gen_dat_model <- function( n_bar = 10,
                           J = 30,
                           p = 0.5,
                           gamma_0 = 0, gamma_1 = 0, gamma_2 = 0,
                           sigma2_u = 0, sigma2_e = 1,
                           alpha = 0 ) {
  # Code (see below) goes here.
}
```

Note our parameters are a mix of *model parameters* (gamma_0, gamma_1, sigma2_e, etc., representing coefficients in regressions, variance terms, etc.) and *design parameters* (n_bar, J, p) that directly inform data generation. We set default arguments (e.g., gamma_0=0) so we can ignore aspects of your DGP that we don't care about later on.

Make the sites. We make the sites first:

```
# generate site sizes
n_min = round( n_bar * (1 - alpha) )
n_max = round( n_bar * (1 + alpha) )
nj <- sample( n_min:n_max, J, replace=TRUE )

# Generate average control outcome and average ATE for all sites
# (The random effects)
u0j = rnorm( J, mean=0, sd=sqrt( sigma2_u ) )

# randomize units within each site (proportion p to treatment)
Zj = ifelse( sample( 1:J ) <= J * p, 1, 0)

# Calculate site intercept for each site
beta_0j = gamma_0 + gamma_1 * Zj + gamma_2 * Zj * (nj-n_bar)/n_bar + u0j
```

Note the line with `sample(1:J) <= J*p`; this is a simple trick to generate treatment and control.

There is also a serious error in the above code; we leave it as an exercise (see below) to find and fix it.

Make the individuals. Then the individuals

```
# Make individual site membership
sid = as.factor( rep( 1:J, nj ) )
dd = data.frame( sid = sid )

# Make individual level tx variables
dd$Z = Zj[ dd$sid ]

# Generate the residuals
N = sum( nj )
e = rnorm( N, mean=0, sd=sqrt( sigma2_e ) )

# Bundle and send out
dd <- mutate( dd,
              sid=as.factor(sid),
              Yobs = beta_0j[sid] + e,
              Z = Zj[ sid ] )
```

The `rep` command will repeat each number, 1, 2... J, the corresponding number of times as listed in nj.

We wrap it all in a function, and when we call it we get:

```
dat <- gen_dat_model( n=5, J=3, p=0.5,
                        gamma_0=0, gamma_1=0.2, gamma_2=0.2,
                        sigma2_u = 0.4, sigma2_e = 1,
                      alpha = 0.5 )

dat
```

```
##    sid Z       Yobs
## 1    1 0  1.3752789
## 2    1 0  0.5802912
## 3    1 0  1.2390264
## 4    1 0 -0.3707643
## 5    1 0  0.3861877
## 6    2 0  1.7129609
## 7    2 0 -0.2930146
## 8    2 0  2.3357674
## 9    2 0  0.9447586
## 10   2 0  3.4317594
## 11   3 1 -1.5382230
## 12   3 1 -2.3565531
## 13   3 1  0.9347565
```

11.4 Analyzing our data

To analyze our data, we use two libraries, the `lme4` package (for multilevel modeling), the `arm` package (which gives us nice access to standard errors, with `se.fixef()`), and `lmerTest` (which gives us p-values for multilevel modeling). We also need the `estimatr` package to get robust SEs with `lm_robust`.

```
library( lme4 )
library( arm )
library( lmerTest )
library( estimatr )
```

We have three analysis functions, which we can put in three different methods:

Multilevel Regression (MLM):

```
analysis_MLM <- function( dat ) {
  M1 = lmer( Yobs ~ 1 + Z + (1|sid),
             data=dat )
  est = fixef( M1 )[["Z"]]
  se = se.fixef( M1 )[["Z"]]
  pv = summary(M1)$coefficients["Z",5]
  tibble( ATE_hat = est, SE_hat = se, p_value = pv )
}
```

Linear Regression with Cluster-Robust Standard Errors (LM):

```
analysis_OLS <- function( dat ) {
  M2 <- lm_robust( Yobs ~ 1 + Z,
            data=dat, clusters=sid )
  est <- M2$coefficients[["Z"]]
  se  <- M2$std.error[["Z"]]
  pv <- M2$p.value[["Z"]]
  tibble( ATE_hat = est, SE_hat = se, p_value = pv )
}
```

Aggregate data (Agg):

```
analysis_agg <- function( dat ) {
  datagg <-
    dat %>%
    group_by( sid, Z ) %>%
    summarise(
      Ybar = mean( Yobs ),
      n = n()
    )

  stopifnot( nrow( datagg ) == length(unique(dat$sid) ) )

  M3 <- lm_robust( Ybar ~ 1 + Z,
                   data=datagg, se_type = "HC2" )
  est <- M3$coefficients[["Z"]]
  se <- M3$std.error[["Z"]]
  pv <- M3$p.value[["Z"]]
  tibble( ATE_hat = est, SE_hat = se, p_value = pv )
}
```

And then a single function that puts all these together:

```
analyze_data = function( dat ) {
  MLM = analysis_MLM( dat )
  LR = analysis_OLS( dat )
  Agg = analysis_agg( dat )

  bind_rows( MLM = MLM, LR = LR, Agg = Agg,
             .id = "method" )
}
```

When we pass a dataset to it, we get a nice table of results that we can evaluate:

```
analyze_data( dat )
```

```
## # A tibble: 3 x 4
##   method ATE_hat SE_hat p_value
##   <chr>    <dbl>  <dbl>   <dbl>
## 1 MLM      -2.12  0.968   0.186
## 2 LR       -2.12  0.492   0.145
## 3 Agg      -2.12  0.492   0.145
```

Each method for analysis is a single line. We record estimated impact, estimated standard error, and a nominal p-value. Note how the `bind_rows()` method can take naming on the fly, and give us a column of `method`, which will be very useful to keep track of what estimated what. We intentionally wrap up our results with a data frame to make later processing of data with the tidyverse package much easier.

11.5 The simulation

So we have our two steps, and the next step of our simulation is to rerun them both a bunch of times. Always start with a single scenario to figure out if your code is working and if your intuition is working.

```
ATE <- 0.30
R <- 1000

one_run <- function( ATE ) {
      dat <- gen_dat_model( n_bar = 200, J=30,
                            gamma_1 = ATE, gamma_2 = 0.5,
                            sigma2_u = 0.20, sigma2_e = 0.80,
                            alpha = 0.75 )
    analyze_data(dat)
```

```
}

tictoc::tic()  # Start the clock!
set.seed( 40404 )
runs <-
  purrr::rerun( R, one_run( ATE ) ) %>%
  bind_rows( .id="runID" )

tictoc::toc()
```

```
## 338.54 sec elapsed
```

We have the individual results of all our methods applied to each generated dataset.

11.6 Analysis of our single scenario

For our single scenario, we can now evaluate how well the estimators did. In particular we have these primary questions:

- Is it biased? (bias)
- Is it precise? (standard error)
- Does it predict well? (RMSE)
- Can we estimate uncertainty well? (i.e., are our estimated SEs about right?)

We systematically go through answering these questions for our initial scenario.

11.6.1 Are the estimators biased?

Bias is with respect to a target estimand. Here we assess whether our estimates are systematically different from the parameter we used to generate the data (this is the ATE parameter). We also calculate the MCSE for the bias using a simple sampling formula.

```
runs %>%
  group_by( method ) %>%
  summarise(
    mean_ATE_hat = mean( ATE_hat ),
    bias = mean( ATE_hat - ATE ),
    SE_bias = sd( ATE_hat - ATE ) / sqrt(R)
  )
```

```
## # A tibble: 3 x 4
##   method mean_ATE_hat    bias SE_bias
##   <chr>         <dbl>   <dbl>   <dbl>
## 1 Agg           0.306 0.00561 0.00531
## 2 LR            0.390 0.0899  0.00580
## 3 MLM           0.308 0.00788 0.00531
```

Linear regression is biased. There is no evidence of bias for Agg or MLM. This is because the linear regression is targeting the person-average average treatment effect. Our data generating process makes larger sites have larger effects, so the person average is going to be higher since those larger sites will count more. The Agg and MLM methods, by contrast, estimate the site-average effect; this is in line with our DGP.

11.6.2 Which method has the smallest standard error?

The true Standard Error is simply how variable the point estimates are, i.e., the standard deviation of the point estimates for a given estimator. The standard error is a measure of how stable our estimates are across datasets that all came from the same data generating process. We calculate the standard error, and also the relative standard error using linear regression as a baseline:

```
true_SE <- runs %>%
  group_by( method ) %>%
  summarise(
    SE = sd( ATE_hat )
  )
true_SE %>%
  mutate( per_SE = SE / SE[method=="LR"] )
```

```
## # A tibble: 3 x 3
##   method    SE per_SE
##   <chr>  <dbl>  <dbl>
## 1 Agg    0.168  0.916
## 2 LR     0.183  1
## 3 MLM    0.168  0.916
```

The other methods appear to have SEs about 8% smaller than Linear Regression

11.6.3 Which method has the smallest Root Mean Squared Error?

So far linear regression is not doing well: it has bias and it also has a larger standard error than the other two. We can assess overall performance by combining

these two quantities with the RMSE:

```
runs %>%
  group_by( method ) %>%
  summarise(
    RMSE = sqrt( mean( (ATE_hat - ATE)^2 ) )
  )
```

```
## # A tibble: 3 x 2
##   method  RMSE
##   <chr>  <dbl>
## 1 Agg    0.168
## 2 LR     0.204
## 3 MLM    0.168
```

RMSE is a way of taking both bias and variance into account, all at once. Here, LR's bias plus increased variability is giving it a higher RMSE. For Agg and MLM, the RMSE is basically the standard error; this makes sense as they are not biased. For LR we see a slight bump to the RMSE, but clearly the standard error dominates the bias term. This is especially the case as RMSE is the square root of the bias and standard errors *squared*; this makes difference between them even more extreme.

11.6.4 Do the methods have correctly estimated standard errors?

To assess this, we can look at the average *estimated* (squared) standard error and compare it to the true standard error. Our standard errors are *inflated* if they are systematically larger than they should be, across the simulation runs. We can also look at how stable our standard error estimates are, by taking the standard deviation of our standard error estimates.

```
runs %>%  group_by( method ) %>%
  summarise(
    SE = sd( ATE_hat ),
    mean_SE_hat = sqrt( mean( SE_hat^2 ) ),
    infl = 100 * mean_SE_hat / SE,
    sd_SE_hat = sd( SE_hat ),
    stability = 100 * sd_SE_hat / SE )
```

```
## # A tibble: 3 x 6
##   method    SE mean_SE_hat  infl sd_SE_hat stability
##   <chr>  <dbl>       <dbl> <dbl>     <dbl>     <dbl>
```

```
## 1 Agg    0.168       0.174  104.    0.0232      13.8
## 2 LR     0.183       0.185  101.    0.0309      16.8
## 3 MLM    0.168       0.174  104.    0.0232      13.8
```

All of the SEs appear to be a bit conservative on average. (3 or 4 percentage points too big).

The stability of the SE-hats is also of interest. The last column shows how variable the standard error estimates are relative to the true standard error. 50% would mean the standard error estimates can easily be off by 50% of the truth, which is not particularly good.

11.6.5 Did we have enough simulation trials?

Finally, we can check our MCSEs for our performance measures to see if we have enough runs to believe these differences:

```
library( simhelpers )
runs$ATE = ATE
runs %>% group_by(method) %>%
    group_modify(
      ~ calc_absolute( ., estimates = ATE_hat, true_param = ATE,
                       perfm_criteria = c("bias","rmse")) )
```

```
## # A tibble: 3 x 6
## # Groups:   method [3]
##   method     K    bias bias_mcse  rmse rmse_mcse
##   <chr>  <int>   <dbl>     <dbl> <dbl>     <dbl>
## 1 Agg     1000 0.00561   0.00531 0.168   0.00461
## 2 LR      1000 0.0899    0.00580 0.204   0.00520
## 3 MLM     1000 0.00788   0.00531 0.168   0.00461
```

We see the MCSEs are small relative to the linear regression bias term and the RMSEs: we simulated enough runs to see these gross trneds.

11.7 Extending to a multifactor simulation

So far, our case study illustrates how multisite simulation (or any simulation) can be constructed by using a model as a recipe for data generation. We have also reinforced our basic message that making functions to wrap parts of simulation substantially eases code construction. But we only investigated a single scenario. How do our findings generalize? When are the different methods

differently appropriate? To answer this, we need to extend to a multifactor simulation to systematically explore trends across contexts for our three estimators. We begin by identifying some questions we might have given our preliminary results.

Regarding bias, in our initial simulation, we noticed that Linear Regression is estimating a person-weighted quantity, and so would be considered biased for the site-average ATE. We might next ask, how much does bias change if we change the site-size by impact relationship?

For precision, we also saw that Linear Regression has a higher standard error. But is this a general finding? When does this occur? Are there contexts where linear regression will do better than the others? Originally we thought aggregation would lose information becuase little sites will have the same weight as big sites, but be more imprecisely estimated. Were we wrong? Or perhaps if site size was even more variable, Agg might do worse and worse.

Finally, the estimated SEs all appeared to be good, although they were rather variable, relative to the true SE. We might then ask, is this always the case? Will the estimated SEs fall apart (e.g., be way too large or way too small, in general) in different contexts?

To answer these questions we need to more systematically explore the space of models. But we have a lot of knobs to turn. In our simulation, we can generate fake cluster randomized data with the following features:

- The treatment impact of the site can vary, and vary with the site size
- We can have sites of different sizes if we want
- We can also vary:
  - the site intercept variance
  - the residual variance,
  - the treatment impact
  - the site size
  - the number of sites, …

We cannot easily vary all of these. We instead reflect on our research questions, speculate as to what is likely to matter, and then consider varying the following:

- Average site size: Does the number of students/site matter?
- Number of sites: Do cluster-robust SEs work with fewer sites?
- Variation in site size: Varying site sizes cause bias or break things?
- Correlation of site size and site impact: Will correlation cause bias?
- Cross site variation: Does the amount of site variation matter?

Even so, we have a problem: How do we index cross site variation? If we simply add more cross site variation, our total variation will increase. If methods deteriorate, we then have a confound: is it the cross site variation causing the problem, or is it the total variation? We therefore want to vary site variation while controlling total variation.

In extending our simulation we will also come across all sorts of concerns such as how to handle convergence issues in the modeling. We also need to think about how to deal with nuisance factors and how to summarize complex simulations. Finally, how do we choose appropriate factors and not become beholden to the parameters in our model?

11.8 Standardization in a data generating process

Given our model, we can generate data by specifying our parameters and variables of $\gamma_0, \gamma_1, \gamma_2, \sigma^2_\epsilon, \sigma^2_u, \bar{n}, \alpha, J, p$.

Now, as discussed above, we want to manipulate within vs. between variation. If we just add more between variation (increase σ^2_u), our overall variation of Y will increase. This will make it hard to think about, e.g., power, since we have confounded within vs. between variation with overall variation (which is itself bad for power). It also impacts interpretation of coefficients. A treatment effect of 0.2 on our outcome scale is "smaller" if there is more overall variation.

To handle this we first (1) Standardize our data and then (2) reparameterize, so we have human-selected parameters that we can interpret that we then *translate* to our list of data generation parameters. This allows us to, for exmaple, operate in standard quantities such as effect size units. It also allows us ot index our DGP with more interpretable parameters such as the Intra-Class Correlation (ICC).

Our model is

$$Y_{ij} = \gamma_0 + \gamma_1 Z_j + \gamma_2 Z_j \left(\frac{n_j - \bar{n}}{\bar{n}} \right) + u_j + \epsilon_{ij}$$

The variance of our control-side outcomes is

$$\begin{aligned} var(Y_{ij}(0)) &= var(\beta_{0j} + \epsilon_{ij}) \\ &= var(\gamma_0 + \gamma_1 Z_j + \gamma_2 Z_j \tilde{n}_j + u_j + \epsilon_{ij}) \\ &= \sigma^2_u + \sigma^2_\epsilon \end{aligned}$$

The effect size of an impact is defined as the impact over the control-side standard deviation. (Sometimes people use the pooled standard deviation, but this

is usually a bad choice if one suspects treatment variation. More treatment variation should not reduce the effect size for the same absolute average impact.)

$$ES = \frac{\gamma_1}{SD(Y|Z_j = 0)} = \frac{\gamma_1}{\sqrt{\sigma_u^2 + \sigma_\epsilon^2}}$$

The way we think about how "big" γ_1 is depends on how much site variation and residual variation there is. But it is also easier to detect effects when the residual variation is small. Effect sizes "standardize out" these sorts of tensions. We can use that.

In particular, we will use the Intraclass Correlation Coeffiicent (ICC), defined as

$$ICC = \frac{\sigma_u^2}{\sigma_\epsilon^2 + \sigma_u^2}.$$

The ICC is a measure of within vs. between variation.

What we then do is first standardized our data, meaning we ensure the control side variance equals 1. Using the above, this means $\sigma_u^2 + \sigma_\epsilon^2 = 1$. It also gives us $ICC = \sigma_u^2$, and $\sigma_\epsilon^2 = 1 - ICC$.

Our two model parameters are now tied together by our single ICC tuning parameter. The core idea is we can now manipulate the aspects of the DGP we want while holding other aspects of the DGP constant. Given our standardized scale, we have dropped a parameter from our set we might want to vary, and ensured varying the other parameter (now the ICC) is varying only one aspect of the DGP, not both. Before, increasing σ_u^2 had two consequences: total variation and relative amount of variation at the school level. Manipulating ICC only does the latter.

Our revised code is then, at the simulation driver level:

```
run_CRT_sim <- function(reps,
                        n_bar = 10, J = 30, p = 0.5,
                        ATE = 0, ICC = 0.4,
                        size_coef = 0, alpha = 0,
                        seed = NULL, aggregate = TRUE) {

  stopifnot( ICC >= 0 && ICC < 1 )

  scat( "Running n=%d, J=%d, ICC=%.2f, ATE=%.2f (%d replicates)\n", n_bar, J, ICC, ATE

  if (!is.null(seed)) set.seed(seed)

  res <-
    purrr::rerun( reps, {
      dat <- gen_dat_model( n_bar = n_bar, J = J, p = p,
```

```
                              gamma_0 = 0, gamma_1 = ATE, gamma_2 = size_coef,
                              sigma2_u = ICC, sigma2_e = 1 - ICC,
                              alpha = alpha )
      analyze_data(dat)
    }) %>%
    bind_rows( .id="runID" )
}
```

Note the `stopifnot`: it is wise to ensure our parameter transforms are all reasonable, so we don't get unexplained errors or strange results.

Also note how we are transforming our ICC parameter into specific other parameters to maintain our effect size interpretation of our simulation. We don't need to modify the `gen_dat_model` method: we are just specifying the constellation of parameters as a function of the parameters we want to directly control in the simulation.

11.9 Making analyze_data() quiet

If we run our simulation when there is little cluster variation, we start getting a lot of warnings from our MLM estimator:

When we scale up to our full simulations, these warnings can become a nuisance. Furthermore, the `lmer` command can sometimes just fails (we believe there is some bug in the optimizer that fails if things are just perfectly wrong). If this was on simulation run 944 out of 1000, we would lose everything! To protect ourselves, we trap messages and warnings as so (see Chapter @(#safe_code) for more on this):

```
quiet_lmer = quietly( lmer )
analyze_data <- function( dat ) {

    # MLM
    dat <- dat
    M1 <- quiet_lmer( Yobs ~ 1 + Z + (1|sid), data=dat )
    message1 = ifelse( length( M1$message ) > 0, 1, 0 )
    warning1 = ifelse( length( M1$warning ) > 0, 1, 0 )

   ...

    # Compile our results
    tibble(
      method = c( "MLM", "LR", "Agg" ),
      ATE_hat = c( est1, est2, est3 ),
```

```
      SE_hat = c( se1, se2, se3 ),
      p_value = c( pv1, pv2, pv3 ),
      message = c( message1, 0, 0 ),
      warning = c( warning1, 0, 0 )
    )
```

We now get a note about the message regarding convergence saved in our results:

```
analyze_data(dat)
```

```
## # A tibble: 3 x 4
##   method ATE_hat SE_hat p_value
##   <chr>    <dbl>  <dbl>   <dbl>
## 1 MLM      -2.12  0.968   0.186
## 2 LR       -2.12  0.492   0.145
## 3 Agg      -2.12  0.492   0.145
```

11.10 Where to compute performance measures: inside vs. outside?

INSIDE (aggregate as you simulate): For each scenario we get a tidy result of our performance measures Less data to store, easier to compartmentalize No ability to add new performance measures on the fly

OUTSIDE (keep all simulation runs): You can dynamically add or change how you calculate performance measures End up with massive amounts of data to store and manipulate

11.11 Run the simulation

Running our simulation is the exact same code as we have used before. Simulations take awhile to run so we save them so we can analyze at our leisure. Here we are storing the individual runs, not the analyzed results!

```
params <-
  cross_df(design_factors) %>%
  mutate(
    reps = 100,
    seed = 20200320 + 1:n()
  )
params$res = pmap(params, .f = one_CRT_sim )
res = params %>% unnest( cols=c(data) )
saveRDS( res, file = "results/simulation_CRT.rds" )
```

11.12 Analyzing our results

Now that we have run our simulation, we turn to our questions listed above, extending our initial findings from our initial scenario to assess trends across our simulation factors.

11.12.1 Checking on convergence issues

First, we explore how often we get a convergence message:

```
res <- readRDS( "results/simulation_CRT.rds" )
res %>%
  group_by( method, ICC ) %>%
  summarise( message = mean( message ) ) %>%
  pivot_wider( names_from = "method", values_from="message" )
```

```
## # A tibble: 5 x 4
##     ICC   Agg    LR      MLM
##   <dbl> <dbl> <dbl>    <dbl>
## 1   0       0     0 0.499
## 2   0.2     0     0 0.0139
## 3   0.4     0     0 0.00311
## 4   0.6     0     0 0.00104
## 5   0.8     0     0 0.000444
```

We see that when the ICC is 0 we get a lot of convergence issues, but as soon as we pull away from 0 it drops off considerably. At this point we might decide to drop those runs with a message or keep them. In this case, we decide to keep (it shouldn't matter much in any case except the ICC = 0 case). We might eventually want to do a separate analysis of the ICC = 0 context to see if the MLM approach actually falls apart, or if it is just throwing error messages.

11.12.2 Calculating standard metrics

Once we have our individual runs for our difference scenarios, we group and calculate our performance metrics for each group.

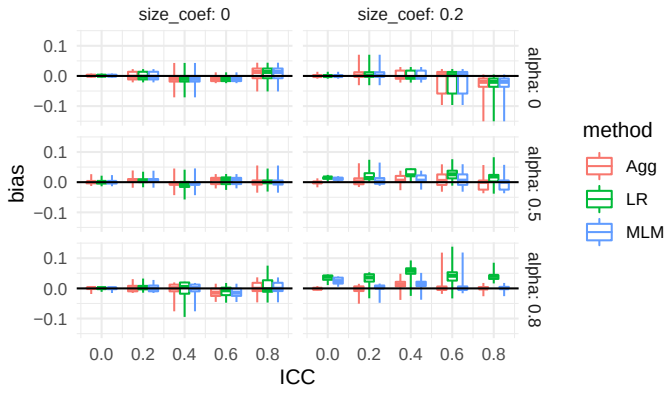
```
sres <-
  res %>%
  group_by( n_bar, J, ATE, size_coef, ICC, alpha, method ) %>%
  summarise(
    bias = mean(ATE_hat - ATE),
    SE = sd( ATE_hat ),
```

```
    RMSE = sqrt( mean( (ATE_hat - ATE )^2 ) ),
    ESE_hat = sqrt( mean( SE_hat^2 ) ),
    SD_SE_hat = sqrt( sd( SE_hat^2 ) ),
    power = mean( p_value <= 0.05 ),
    R = n(),
    .groups = "drop"
  )
```

11.12.3 Bias analysis

As a first step to understanding bias, we might bundle our results by ICC. In this code we are making groups of method by ICC level so we get side-by-side boxplots for each ICC level considered:

```
ggplot( sres, aes( ICC, bias, col=method, group=paste0(ICC,method) ) ) +
  facet_grid( alpha ~ size_coef, labeller = label_both ) +
  geom_boxplot(coef = Inf) +
  geom_hline( yintercept = 0 ) +
  theme_minimal() +
  scale_x_continuous( breaks = unique( sres$ICC) )
```



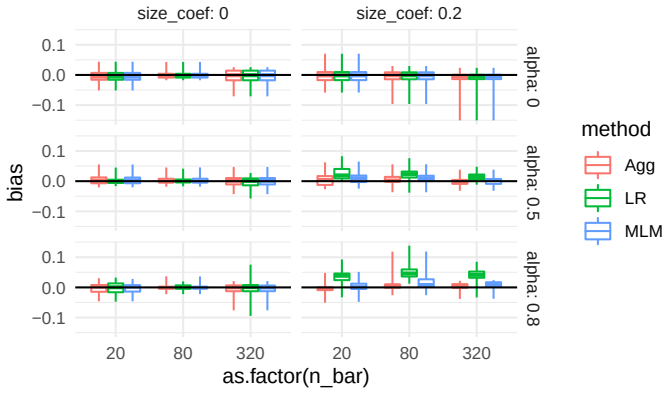
Each box is a collection of simulation trials. E.g., for `ICC = 0.6`, `size_coef = 0.2`, and `alpha = 0.8` we have 9 scenarios representing the varying level 1 and level 2 sample sizes:

```
filter( sres, ICC == 0.6, size_coef == 0.2,
        alpha == 0.8, method=="Agg" ) %>%
  dplyr::select( n_bar:alpha, bias )
```

```
## # A tibble: 9 x 7
##   n_bar     J   ATE size_coef   ICC alpha     bias
##   <dbl> <dbl> <dbl>     <dbl> <dbl> <dbl>    <dbl>
## 1    20     5   0.2       0.2   0.6   0.8 -0.00452
## 2    20    20   0.2       0.2   0.6   0.8 -0.0182
## 3    20    80   0.2       0.2   0.6   0.8 -0.00921
## 4    80     5   0.2       0.2   0.6   0.8  0.119
## 5    80    20   0.2       0.2   0.6   0.8  0.00210
## 6    80    80   0.2       0.2   0.6   0.8  0.00641
## 7   320     5   0.2       0.2   0.6   0.8 -0.00182
## 8   320    20   0.2       0.2   0.6   0.8 -0.00219
## 9   320    80   0.2       0.2   0.6   0.8  0.00632
```

We are seeing a few outliers for some of the boxplots, suggesting that there are other factors driving bias. We could try bundling along different aspects to see:

```
ggplot( sres, aes( as.factor(n_bar), bias, col=method, group=paste0(n_bar,method) ) ) +
  facet_grid( alpha ~  size_coef, labeller = label_both ) +
  geom_boxplot(coef = Inf) +
  geom_hline( yintercept = 0 ) +
  theme_minimal()
```

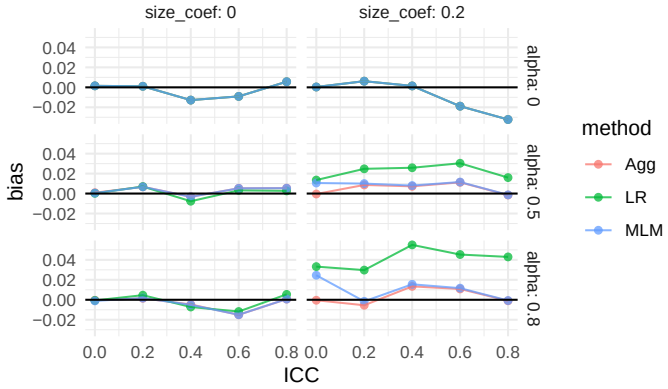


No progress there. Perhaps it is instability or MCSE.

The boxplots are hard for seeing trends. Instead of bundling, we can therefore aggregate:

```
ssres <-
  sres %>%
  group_by( ICC, method, alpha, size_coef ) %>%
  summarise( bias = mean( bias ) )
```

```
ggplot( ssres, aes( ICC, bias, col=method ) ) +
  facet_grid( alpha ~  size_coef, labeller = label_both ) +
  geom_point( alpha=0.75 ) +
  geom_line( alpha=0.75 ) +
  geom_hline( yintercept = 0 ) +
  theme_minimal()
```



This shows that site variation leads to greater bias, but only if the coefficient for size is nonzero. We also see that all the estimators must be the same if site variation is 0, with the overplotted lines on the top row of the figure.

11.13 Exercises

1. What is the variance of the outcomes generated by our model if there is no treatment effect? (Try simulating data to check!) What other quick checks can you make on your DGP to make sure it is working?

2. In gen_dat_model we have the following line of code to generate the number of individuals per site.

```
nj <- sample( n_min:n_max, J,
                  replace=TRUE )
```

This code has an error. Generate a variety of datasets where you vary `n_min`, `n_max` and `J` to discover the error. Then repair the code. Checking your data generating process across a range of scenarios is extremely important.

2. Extend the data generating process to include individual level covariates?

3. As foreground to the following chapters, can you explore multiple scenarios to see if the trends are common? First write a function that takes a set of parameters and runs the entire simulation and returns the results as a small dataframe. Then use code like this to make a graph of some result measure as a function of a varying parameter (you pick which parameter you wish to vary):

```
vals = seq( start, stop, length.out = 5 )
res = map_df( vals, my_simulation_function,
              par1 = val1, par2 = val2, etc )
```

Chapter 12

Error trapping and other headaches

If you have an advanced estimator, or are relying on some package, it is quite possible that every so often your estimate will trigger an error, or give you a NA result, or something similarly bad. More innocuous, you might have estimators that can generate warnings; if you run 10,000 trials, that can add up to a lot of warnings which can be overwhelming to sort through. In some cases, these warnings might be coupled with the estimator returning a very off result; it is unclear, in this case, whether we should include that result in our overall performance measures for that estimator. After all, it tried to warn us!

In this section, we talk about some ways to make your simulations safe and robust, and also discuss some ways to track warnings and include them in peformance measures.

12.1 Safe code

Sometimes if you write a function that does a lot of complex things with uncertain objects – i.e. consider a complex estimator using an unstable package not of your own creation – you can run into trouble where you have a intermittent error.

This can really be annoying; consider the case where your simulation crashes on 1 out of 500 chance: if you run 1000 simulation trials, your program will likely not make it to the end, thus wasting all of your time. You can write code that can, instead of stopping when it reaches an error, trap the error and move on with the next simulation trial.

To illustrate, consider the following broken function:

```
my_complex_function = function( param ) {

    vals = rnorm( param, mean = 0.5 )
    if ( sum( vals ) > 5 ) {
        broken_code( 4 )
    } else {
        sqrt( sum( vals ) )
    }
}
```

We run it like so:

```
my_complex_function( 1 )
```

```
## [1] 1.233594
```

```
my_complex_function( 7 )
```

```
## Error in broken_code(4): could not find function "broken_code"
```

```
resu = rerun( 20, my_complex_function( 7 ) )
```

```
## Warning in sqrt(sum(vals)): NaNs produced

## Error in broken_code(4): could not find function "broken_code"
```

Oh no! Our function crashes sometimes. To trap the errors we use the purrr package to make a "safe" function as so:

```
my_safe_function = safely( my_complex_function )
my_safe_function( 7 )
```

```
## $result
## [1] 1.340666
##
## $error
## NULL
```

There are other function wrappers in this family, such as "possibly":

```
my_possible_function = possibly( my_complex_function,
                                 otherwise = NA )
my_possible_function( 7 )
```

```
## [1] 1.89293
```

```
as.numeric( rerun( 10, my_possible_function(7) ) )
```

```
## Warning in sqrt(sum(vals)): NaNs produced
```

```
##  [1] 0.2912577 0.9354464       NaN 2.0169705 2.1757312        NA        NA
##  [8]        NA        NA 1.8955811
```

Finally, we can use "quietly" to make warnings and stuff go away:

```
my_quiet_function = quietly( my_complex_function )

my_quiet_function( 1 )
```

```
## $result
## [1] NaN
##
## $output
## [1] ""
##
## $warnings
## [1] "NaNs produced"
##
## $messages
## character(0)
```

This can be especially valuable to control massive amounts of printout in a simulation. If you have lots of extraneous printout, it can slow down the execution of your code far more than you might think.

12.1.1 What to do with warnings

Sometimes our analytic strategy might give some sort of warning (or fail altogether). For example, from the cluster randomized experiment case study we have:

```
set.seed(101012)  # (I picked this to show a warning.)
dat = gen_dat_model( J = 50, n_bar = 100, sigma2_u = 0 )
mod <- lmer( Yobs ~ 1 + Z + (1|sid), data=dat )
```

```
## boundary (singular) fit: see help('isSingular')
```

We have to make a deliberate decision as to what to do about this:

- Keep these "weird" trials?
- Drop them?

If you decide to drop them, you should drop the entire simulation iteration including the other estimators, even if they worked fine! If there is something particularly unusual about the dataset, then dropping for one estimator, and keeping for the others that maybe didn't give a warning, but did struggle to estimate the estimand, would be unfair: in the final performance measures the estimators that did not give a warning could be being held to a higher standard, making the comparisons between estimators biased.

If your estimators generate warnings, you should calculate the rate of errors or warning messages as a performance measure. Especially if you drop some trials, it is important to see how often things are acting pecularly.

The main tool for doing this is the `quietly()` function:

```
quiet_lmer = quietly( lmer )
qmod <- quiet_lmer( Yobs ~ 1 + Z + (1|sid), data=dat )
qmod
```

```
## $result
## Linear mixed model fit by REML ['lmerModLmerTest']
## Formula: ..1
##    Data: ..2
## REML criterion at convergence: 6485.293
## Random effects:
##  Groups   Name        Std.Dev.
##  sid      (Intercept) 0.0000
##  Residual             0.9879
## Number of obs: 2302, groups:  sid, 50
## Fixed Effects:
## (Intercept)            Z
##    -0.03143      0.03433
## optimizer (nloptwrap) convergence code: 0 (OK) ; 0 optimizer warnings; 1 lme4 warni
##
```

```
## $output
## [1] ""
##
## $warnings
## character(0)
##
## $messages
## [1] "boundary (singular) fit: see help('isSingular')\n"
```

You then might have, in your analyzing code:

```
analyze_data <- function( dat ) {

    M1 <- quiet_lmer( Yobs ~ 1 + Z + (1|sid), data=dat )
    message1 = ifelse( length( M1$message ) > 0, 1, 0 )
    warning1 = ifelse( length( M1$warning ) > 0, 1, 0 )

    # Compile our results
    tibble( ATE_hat = coef(M1)["Z"],
            SE_hat = se.coef(M1)["Z"],
            message = message1,
            warning = warning1 )
}
```

Now you have your primary estimates, and also flags for whether there was a convergence issue. In the analysis section you can then evaluate what proportion of the time there was a warning or message, and then do subset analyses to those simulation trials where there was no such warning.

12.2 Saving files and results

Always save your simulation results to a file. Simulations are painful and time consuming to run, and you will invariably want to analyze the results of them in a variety of different ways, once you have looked at your preliminary analysis. We advocate saving your simulation as soon as it is complete. But there are some ways to do better than that, such as saving as you go. This can protect you if your simulation occasionally crashes, or if you want to rerun only parts of your simulation for some reason.

12.2.1 Saving simulations as you go

If you are not sure you have time to run your entire simulation, or you think your computer might crash half way through, or something similar, you can save

each chunk you run as you go, in its own file. You then stack those files at the end to get your final results. With clever design, you can even then selectively delete files to rerun only parts of your larger simulation—but be sure to rerun everything from scratch before you run off and publish your results, to avoid embarrassing errors.

Here, for example, is a script from a research project examining how one might use post-stratification to improve the precision of an IV estimate. This is the script that runs the simulation. Note the sourcing of other scripts that have all the relevant functions; these are not important here. Due to modular programming, we can see what this script does, even without those detail.

```
source( "pack_simulation_functions.R" )

if ( !file.exists("results/frags" ) ) {
    dir.create("results/frags")
}

# Number of simulation replicates per scenario
R = 1000

# Do simulation breaking up R into this many chunks
M_CHUNK = 10

###### Set up the multifactor simulation #######

# chunkNo is a hack to make a bunch of smaller chunks for doing parallel more
# efficiently.
factors = expand_grid( chunkNo = 1:M_CHUNK,
                       N = c( 500, 1000, 2000 ),
                       pi_c = c( 0.05, 0.075, 0.10 ),
                       nt_shift = c( -1, 0, 1 ),
                       pred_comp = c( "yes", "no" ),
                       pred_Y = c( "yes", "no" ),
                       het_tx = c( "yes", "no" ),
                       sd0 = 1
                       )
factors <- factors %>% mutate(
    reps = R / M_CHUNK,
    seed = 16200320 + 1:n()
)
```

This generates a data frame of all our factor combinations. This is our list of "tasks" (each row of factors). These tasks have repeats: the "chunks" means we do a portion of each scenario, as specified by our simulation factors, as a process. This would allow for greater parallelization (e.g., if we had more cores),

and also lets us save our work without finishing an entire scenario of, in this case, 1000 iterations.

To set up our simulation we make a little helper method to do one row. With each row, once we have run it, we save it to disk. This means if we kill our simulation half-way through, most of the work would be saved. Our function is then going to either do the simulation (and save the result to disk immediately), or, if it can find the file with the results from a previous run, load those results from disk:

```
safe_run_sim = safely( run_sim )
file_saving_sim = function( chunkNo, seed, ... ) {
    fname = paste0( "results/frags/fragment_", chunkNo, "_", seed, ".rds" )
    res = NA
    if ( !file.exists(fname) ) {
        res = safe_run_sim( chunkNo=chunkNo, seed=seed, ... )
        saveRDS(res, file = fname )
    } else {
        res = readRDS( file=fname )
    }
    return( res )
}
```

Note how we wrap our core `run_sim` method in `safely`; it was crashing very occasionally, and so to make the code more robust, we wrapped it so we could see any error messages.

We next run the simulation. We shuffle the rows of our task list so that which process gets what task is randomized. If some tasks are much longer (e.g., due to larger sample size) then this will get balanced out across our processes.

We have an `if-then` structure to easily switch between parallel and nonparallel code. This makes debugging easier: when running in parallel, stuff printed to the console does not show until the simulation is over. Plus it would be all mixed up since multiple processes are working simultaneously.

This overall structure allows the researcher to delete one of the "fragment" files from the disk, run the simulation code, and have it just do one tiny piece of the simulation. This means the researcher can insert a `browser()` command somewhere inside the code, and debug the code, in the natural context of how the simulation is being run.

```
# Shuffle the rows so we run in random order to load balance.
factors = sample_n(factors, nrow(factors) )

if ( TRUE ) {
    # Run in parallel
```

```
    parallel::detectCores()

    library(future)
    library(furrr)

    #plan(multiprocess) # choose an appropriate plan from future package
    #plan(multicore)
    plan(multisession, workers = parallel::detectCores() - 2 )

    factors$res <- future_pmap(factors, .f = file_saving_sim,
                           .options = furrr_options(seed = NULL),
                           .progress = TRUE )

} else {
  # Run not in parallel, used for debugging
  factors$res <- pmap(factors, .f = file_saving_sim )
}

tictoc::toc()
```

Our method cleverly loads files in, or generates them, for each chunk. The seed setting ensures reproducibility. Once we are done, we need to clean up our results:

```
sim_results <-
    factors %>%
    unnest(cols = res)

# Cut apart the results and error messages
sim_results$sr = rep( c("res","err"), nrow(sim_results)/2)
sim_results = pivot_wider( sim_results, names_from = sr, values_from = res )

saveRDS( sim_results, file="results/simulation_results.rds" )
```

12.2.2 Dynamically making directories

If you are generating a lot of files, then you should put them somewhere. But where? It is nice to dynamically generate a directory for your files on fly. One way to do this is to write a function that will make any needed directory, if it doesn't exist, and then put your file in that spot. For example, you might have your own version of `write_csv` as:

```
my_write_csv <- function( data, path, file ) {

  if ( !dir.exists( here::here( path ) ) ) {
    dir.create( here::here( path ), recursive=TRUE )
  }
  write_csv( data, paste0( path, file ) )
}
```

This will look for a path (starting from your R Project, by taking advantage of the `here` package), and put your data file in that spot. If the spot doesn't exist, it will make it for you.

12.2.3 Loading and combining files of simulation results

Once your simulation files are all generated, the following code will stack them all into a giant set of results, assuming all the files are themselves data frames stored in RDS objects. This function will try and stack all files found in a given directory; for it to work, you should ensure there are no other files stored there.

```
load.all.sims = function( filehead="results/" ) {

  files = list.files( filehead, full.names=TRUE)

  res = map_df( files, function( fname ) {
    cat( "Reading results from ", fname, "\n" )
    rs = readRDS( file = fname )
    rs$filename = fname
    rs
  })
  res
}
```

You would use as so:

```
results = load.all.sims( filehead="raw_results/" )
```

Chapter 13

Designing the multifactor simulation experiment

So far, we've created code that will give us results for a single combination of parameter values. In practice, simulation studies typically examine a range of different values, including varying the level of the true parameter values and perhaps also varying sample sizes. Let's now look at the remaining piece of the simulation puzzle: the study's experimental design.

Simulation studies often take the form of **full factorial** designed experiments. In full factorials, each factor (a particular knob a researcher might turn to change the simulation conditions) is varied across multiple levels, and the design includes *every* possible combination of the levels of every factor. One way to represent such a design is as a list of factors and levels.

For example, for the Cronbach alpha simulation, we might want to vary:

- the true value of alpha, with values ranging from 0.1 to 0.9;
- the degrees of freedom of the multivariate t distribution, with values of 5, 10, 20, or 100;
- the sample size, with values of 50 or 100; and
- the number of items, with values of 4 or 8.

Here is code that implements this design, using 500 replications per condition:

```
# first express the simulation parameters as a list of factors, each
# factor having a list of values to explore.
design_factors <- list(
  n = c(50, 100),
  p = c(4, 8),
```

```
  alpha = seq(0.1, 0.9, 0.1),
  df = c(5, 10, 20, 100)
)

params <- cross_df(design_factors)
params$iterations <- 500
params$seed <- 20170405 + 1:nrow(params)
```

This gives us a $2 \times 2 \times 9 \times 4$ factorial design:

```
lengths(design_factors)
```

```
##     n     p alpha    df
##     2     2     9     4
```

The `params` data frame is a representation of the full experimental design:

```
params
```

```
## # A tibble: 144 x 6
##        n     p alpha    df iterations     seed
##    <dbl> <dbl> <dbl> <dbl>      <dbl>    <dbl>
##  1    50     4   0.1     5        500 20170406
##  2   100     4   0.1     5        500 20170407
##  3    50     8   0.1     5        500 20170408
##  4   100     8   0.1     5        500 20170409
##  5    50     4   0.2     5        500 20170410
##  6   100     4   0.2     5        500 20170411
##  7    50     8   0.2     5        500 20170412
##  8   100     8   0.2     5        500 20170413
##  9    50     4   0.3     5        500 20170414
## 10   100     4   0.3     5        500 20170415
## # ... with 134 more rows
```

We see we have a total of 144 cells, each cell corresponding to a simulation scenario to explore.

13.1 Choosing parameter combinations

We've seen how to create a set of experimental conditions, but how do we go about choosing parameter values to examine? Choosing parameters is a central part of good simulation design because the primary limitation of simulation

studies is always their *generalizability*. On the one hand, it's difficult to extrapolate findings from a simulation study beyond the set of simulation conditions that were examined. On the other hand, it's often difficult or impossible to examine the full space of all possible parameter values, except for very simple problems. Even in the Cronbach alpha simulation, we've got four factors, and the last three could each take an infinite number of different levels, in theory. How can we come up with a defensible set of levels to examine?

The choice of simulation conditions needs to be made in the context of the problem or model that you're studying, so it's a bit difficult to offer valid, decontextualized advice. We can provide a couple of observations all the same:

1. For research simulations, it often is important to be able to relate your findings to previous research. This suggests that you should select parameter levels to make this possible, such as by looking at sample sizes similar to those examined in previous studies. That said, previous simulation studies are not always perfect (actually, there's a lot of really crummy ones out there!), and so this should not be your sole guide or justification.

2. Generally, it is better to err on the side of being more comprehensive. You learn more by looking at a broader range of conditions, and you can always boil down your results to a more limited set of conditions for purposes of presentation.

3. It is also important to explore breakdown points (e.g., what sample size is too small for a method to work?) rather than focusing only on conditions where a method might be expected to work well. Pushing the boundaries and identifying conditions where estimation methods break will help you to provide better guidance for how the methods should be used in practice.

An important point regarding (2) is that you can be more comprehensive and then have fewer replications per scenario. For example, say you were planning on doing 1000 simulations per scenario, but then you realize there is some new factor that you don't think matters, but that you believe other researchers will worry about. You could add in that factor, say with four levels, and then do 250 simulations per scenario. The total work remains the same.

When analyzing the final simulation you can then first verify you do not see trends along this new factor, and then marganalize out the factor in your summaries of results. Marganalizing out a factor (i.e., averaging your performance metrics across the additional factor) is a powerful technique to make a claim about how your methods work *on average* across a *range* of scenarios, rather than for a specific scenario.

13.2 Using pmap to run multifactor simulations

To run simulations across all of our factor combinations, we are going to use a very useful method in the `purrr` package called `pmap()`. `pmap()` marches down a set of lists, running a function on each p-tuple of elements, passing them as parameters. It returns a list of the results of this sequence of function calls.

```
my_function <- function( a, b, theta, scale ) {
    scale * (a + theta*(b-a))
}

args = list( a = 1:3,
             b = 5:7,
             theta = c(0.2, 0.3, 0.7) )
purrr::pmap_dbl(  args, my_function, scale = 10 )
```

```
## [1] 18 32 58
```

One important note is the variable names for the lists being iterated over must correspond exactly to function arguments of the called function. Extra parameters can be passed after the function name; these will be held constant, and passed to each function call.

As we see above, `pmap()` has variants such as `_dbl` or `_df` just like the `map()` and `map2()` methods. These variants will automatically stack or convert the list of things returned into a tidier collection (for `_dbl` it will convert to a vector of numbers, for `_df` it will stack to make a large dataframe, assuming each thing returned is a little dataframe).

Ok, but this doesn't quite look like what we want: our factors are stored as a dataframe, not three lists. This is where R gets interesting: data frames are lists of vectors. Witness:

```
args[[2]]
```

```
## [1] 5 6 7
```

```
a_df = as.data.frame(args)
a_df
```

```
##   a b theta
## 1 1 5   0.2
## 2 2 6   0.3
## 3 3 7   0.7
```

```
a_df[[2]]
```

```
## [1] 5 6 7
```

```
purrr::pmap_dbl( a_df, my_function, scale = 10)
```

```
## [1] 18 32 58
```

Note how we can pass `a_df` to `pmap`, and have it do exactly what it did with the lists. This is because the way R stores a dataframe is as a list of vectors or lists (with each of the vectors or lists having the exact same length). This works beautifully with `pmap()`.

All of this means `pmap()` can run a specified function on each row of a dataset. Continuing the Cronback Alpha simulation from above, we would have the following:

This allows us to call our `run_alpha_sim()` method for each row of our list of scenarios we want to explore. When we do, we can also store the results **as a new variable in the same dataset**:

```
sim_results <- params
sim_results$res <- pmap(params, .f = run_alpha_sim)
```

When we do this, we will be creating a **list-column**, where each observation is a little dataset:

```
sim_results
```

```
## # A tibble: 144 x 7
##        n     p alpha    df iterations     seed res
##    <dbl> <dbl> <dbl> <dbl>      <dbl>    <dbl> <list>
##  1    50     4   0.1     5        500 20170406 <df [4 x 3]>
##  2   100     4   0.1     5        500 20170407 <df [4 x 3]>
##  3    50     8   0.1     5        500 20170408 <df [4 x 3]>
##  4   100     8   0.1     5        500 20170409 <df [4 x 3]>
##  5    50     4   0.2     5        500 20170410 <df [4 x 3]>
##  6   100     4   0.2     5        500 20170411 <df [4 x 3]>
##  7    50     8   0.2     5        500 20170412 <df [4 x 3]>
##  8   100     8   0.2     5        500 20170413 <df [4 x 3]>
##  9    50     4   0.3     5        500 20170414 <df [4 x 3]>
## 10   100     4   0.3     5        500 20170415 <df [4 x 3]>
## # ... with 134 more rows
```

Each element in our list column is the little summary of our simulation results for that scenario. Here is the third scenario, for example:

```
sim_results$res[[3]]
```

```
##         criterion         est        MCSE
## 1      alpha bias -0.07810538 0.015587246
## 2      alpha RMSE  0.35684535 0.002040271
## 3 V relative bias  0.40563277 0.004957216
## 4        coverage  0.83000000 0.009746794
```

We finally use `unnest()` to expand the `res` variable, replicating the values of the main variables once for each row in the nested dataset:

```
library(tidyr)
sim_results <- unnest(sim_results, cols = res)
sim_results
```

```
## # A tibble: 576 x 9
##        n     p alpha    df iterations     seed criterion           est     MCSE
##    <dbl> <dbl> <dbl> <dbl>      <dbl>    <dbl> <chr>             <dbl>    <dbl>
##  1    50     4   0.1     5        500 20170406 alpha bias      -0.0671 0.0135
##  2    50     4   0.1     5        500 20170406 alpha RMSE       0.309  0.000536
##  3    50     4   0.1     5        500 20170406 V relative bias  0.601  0.00172
##  4    50     4   0.1     5        500 20170406 coverage         0.854  0.00975
##  5   100     4   0.1     5        500 20170407 alpha bias      -0.0469 0.0105
##  6   100     4   0.1     5        500 20170407 alpha RMSE       0.239  0.000480
##  7   100     4   0.1     5        500 20170407 V relative bias  0.462  0.00165
##  8   100     4   0.1     5        500 20170407 coverage         0.804  0.00975
##  9    50     8   0.1     5        500 20170408 alpha bias      -0.0781 0.0156
## 10    50     8   0.1     5        500 20170408 alpha RMSE       0.357  0.00204
## # ... with 566 more rows
```

We can put all of this together in a a tidy workflow as follows:

```
sim_results <-
  params %>%
  mutate(res = pmap(., .f = run_sim)) %>%
  unnest(cols = res)
```

If we wanted to use parallel processing (more on this later) we can also simply use the `simhelpers` package (the following code is auto-generated by the `create_skeleton()` method as well):

```
plan(multisession) # choose an appropriate plan from the future package
evaluate_by_row(params, run_alpha_sim)
```

13.3 Keeping things organized and the source command

Once you have your multifactor simulation, if it is a particularly complex one, you will have three general collections of code:

- Code for generating data
- Code for analyzing data
- Code for running a single simulation scenario

If each of these pieces is large and complex, you might consider putting them in three different `.R` files. Then, in your primary simulation, you would source these files. E.g.,

```
source( "pack_data_generators.R" )
source( "pack_estimators.R" )
source( "pack_simulation_support.R" )
```

You might also just have the `pack_simulation_support.R` source the other two files, and then source the single simulation support file.

One reason for doing this is you can then have testing code in each of your files, testing each of your components. When you are not focused on that component, you don't have to look at that testing code.

Another is you can have a variety of data generators, forming a library of options. You can then create different simulations that use different pieces, in a larger project.

For example, in one recent simulation project on estimators for an Instrumental Variable analysis, we had several different data generators for generating different types of compliance patterns (IVs are often used to handle noncompliance in randomized experiments). Our file then had several methods:

```
> ls()
[1] "describe_sim_data"  "make_dat"           "make.dat.1side"     "make.dat.1side.old" "make.dat
[6] "make.dat.simple"    "make.dat.tuned"     "rand.exp"           "summarize_sim_data"
```

The describe and summarize methods printed various statistics about a sample dataset; these are used to debug and understand how the generated data looks.

We also had a variety of different DGP methods because we had different versions that came up as we were trying to chase down errors in our estimators and understand strange behavior.

Putting the estimators in a different file also had a nice additional purpose: we also had an applied data example in our work, and we could simply source that file and use those estimators on our actual data. This ensured our simulation and applied analysis were perfectly aligned in terms of the estimators we were using. Also, as we debugged our estimators and tweaked them, we immediately could re-run our applied analysis to update those results with minimal effort.

Modular programming is key.

13.4 Analyzing results from a multifactor experiment

We can group by our simulation factors and calculate all our performance metrics at once. For example, here is the code for calculating performance measures across our simulation for cluster randomized experiments:

```
res <- readRDS( file = "results/simulation_CRT.rds" )

sres <-
  res %>%
  group_by( n_bar, J, ATE, size_coef, ICC, alpha, method ) %>%
  summarise(
    bias = mean(ATE_hat - ATE),
    SE = sd( ATE_hat ),
    RMSE = sqrt( mean( (ATE_hat - ATE )^2 ) ),
    ESE_hat = sqrt( mean( SE_hat^2 ) ),
    SD_SE_hat = sqrt( sd( SE_hat^2 ) ),
    power = mean( p_value <= 0.05 ),
    R = n(),
    .groups = "drop"
  )
sres
```

```
## # A tibble: 810 x 14
##    n_bar     J   ATE size_coef   ICC alpha method     bias    SE  RMSE ESE_hat
##    <dbl> <dbl> <dbl>     <dbl> <dbl> <dbl> <chr>     <dbl> <dbl> <dbl>   <dbl>
##  1    20     5   0.2         0   0     0   Agg     0.00806 0.200 0.200   0.197
##  2    20     5   0.2         0   0     0   LR      0.00806 0.200 0.200   0.197
##  3    20     5   0.2         0   0     0   MLM     0.00806 0.200 0.200   0.234
##  4    20     5   0.2         0   0     0.5 Agg     0.0265  0.211 0.213   0.217
```

```
##  5    20     5   0.2         0   0     0.5 LR      0.0214  0.209 0.210   0.209
##  6    20     5   0.2         0   0     0.5 MLM     0.0234  0.208 0.209   0.244
##  7    20     5   0.2         0   0     0.8 Agg    -0.0186  0.239 0.239   0.224
##  8    20     5   0.2         0   0     0.8 LR     -0.0114  0.226 0.226   0.200
##  9    20     5   0.2         0   0     0.8 MLM    -0.0153  0.227 0.227   0.246
## 10    20     5   0.2         0   0.2   0   Agg    -0.00585 0.446 0.445   0.455
## # ... with 800 more rows, and 3 more variables: SD_SE_hat <dbl>, power <dbl>,
## #   R <int>
```

But then what? We have 6.75×10^4 different scenarios across our factors (with three rows per scenario, one for each method). How can we visualize and understand trends across this complex domain.

There several techniques for summarizing across the data that one might use.

13.4.1 Bundling

As a first step, we might bundle the simulations by the primary factors of interest. We would then plot these bundles as box plots to see central tendency along with variation. With bundling, we would need a good number of simulation runs per scenario, so that the MCSE in the performance measures does not make our boxplots look substantially more variable than the truth.

13.4.2 Aggregation

With aggregation, we average over some of the factors, collapsing our simulation results down to fewer moving parts. This is better than having not had those factors in the first place! Averaging over a factor is a more general answer than having not varied the factor at all.

For example, if we average across ICC and site variation, and see how the methods change performance as a function of J, we would know that this is a general trend across a range of scenarios defined by different ICC and site variation levels. Our conclusions would then be more general than if we picked a single ICC and amount of site variation: in this latter case we would not know if we would see our trend more broadly.

Also, with aggegation, we can have a smaller number of replications per factor combination. The averaging will, in effect, give a lot more reps per aggregated performance measure.

Aggregate, grouping only by the factors used in the plot.

A caution with aggregation is that it can be deceitful if you have scaling issues or extreme outliers. With bias, our scale is fairly well set, so we are good!

13.4.3 Regression Summarization

One can treat the simulation results as a dataset in its own right. In this case we can regress a performance measure against the methods and various factor levels to get "main effects" of how the different levels impact performance holding the other levels constant.

13.4.4 Focus on subset, kick rest to supplement

Frequently researchers might simply filter the simulation results to a single factor level for some nuisance parameter. For example, we might examine ICC of 0.20 only, as this is a "reasonable" value given substance matter knowledge. We would then consider the other levels as a "sensitivity" analysis vaguely alluded to in our main report and placed elsewhere, like in an online supplemental appendix.

It would be our job, in this case, to verify that our reported findings on the main results indeed were echoed in our other, set-aside, simulation runs.

Chapter 14

Case study: A simulation to compare different estimators

In this case study we examine a simulation where we wish to compare different forms of estimator for estimating the same thing. We still generate data, evaluate it, and see how well our evaluation works. The difference is we now evaluate it multiple ways, storing how the different ways work.

For our simple working example we are going to compare estimation of the center of a symmetric distribution via mean, trimmed mean, and median (so the mean and median are the same). These are the three estimation strategies that we might be comparing in a paper (pretend we have "invented" the trimmed mean and want to demonstrate its utility).

We are, as usual, going to break building this simulation evaluation down into lots of functions to show the general framework. This framework can readily be extended to more complicated simulation studies. This case study illustrates how methodologists might compare different strategies for estimation, and is closest to what we might see in the "simulation" section of a stats paper.

14.1 The data generating process

For our data-generation function we will use the scaled t-distribution so the standard deviation will always be 1 but we will have different fatness of tails (high chance of outliers):

```
gen.data = function( n, df0 ) {
    rt( n, df=df0 ) / sqrt( df0 / (df0-2) )
}
```

The variance of a t is $df/(df-2)$, so if we divide our observations by the square root of this, we will standardize them so they have unit variance. See, the standard deviation is 1 (up to random error, and as long as df0 > 2)!:

```
sd( gen.data( 100000, df0 = 3 ) )
```

```
## [1] 0.9974364
```

(Normally our data generation code would be a bit more fancy.)

We next define the parameter we want (in our case this is the mean, is what we are trying to estimate):

```
mu = 0
```

14.2 The data analysis methods

We then write a function that takes data and uses all our different estimators on it. We return a data frame of the three estimates, with each row being one of our estimators. This is useful if our estimators return an estimate and a standard error, for example.

```
analyze.data = function( data ) {
    mn = mean( data )
    md = median( data )
    mn.tr = mean( data, trim=0.1 )
    data.frame( estimator = c( "mean", "trim.mean", "median" ),
                estimate = c( mn, mn.tr, md ) )
}
```

Let's test:

```
dt = gen.data( 100, 3 )
analyze.data( dt )
```

```
##   estimator     estimate
## 1      mean -0.006977732
## 2 trim.mean  0.041118938
## 3    median  0.035254949
```

Note that we have bundled our multiple methods into a single function. With complex methods we generally advocate a separate function for each method, but sometimes for a target simulation having a host of methods wrapped in a single function is clean and tidy code.

Also note the three lines of output for our returned value. This long-form output will make processing the simulation results easier. That being said, returning in wide format is also completely legitimate.

14.3 The simulation itself

To evaluate, do a bunch of times, and assess results. Let's start by looking at a specific case. We generate 1000 datasets of size 10, and estimate the center using our three different estimators.

```
raw.exps <- replicate( 1000, {
    dt = gen.data( n=10, df0=5 )
    analyze.data( dt )
}, simplify = FALSE )
raw.exps = bind_rows( raw.exps, .id = "runID" )
```

Note how our `.id` argument gives each simulation run an ID. This can be useful to see how the estimators covary.

We now have 1000 estimates for each of our estimators:

```
head( raw.exps )
```

```
##   runID estimator   estimate
## 1     1      mean -0.4413881
## 2     1 trim.mean -0.2796097
## 3     1    median -0.4573288
## 4     2      mean  0.2612034
## 5     2 trim.mean  0.2202813
## 6     2    median  0.2510697
```

14.4 Calculating performance measures for all our estimators

We then want to assess estimator performance for each estimator. We first write a function to calculate what we want from 1000 estimates:

```
estimator.quality = function( estimates, mu ) {
    RMSE = sqrt( mean( (estimates - mu)^2 ) )
    bias = mean( estimates - mu )
    SE = sd( estimates )
    data.frame( RMSE=RMSE, bias=bias, SE=SE )
}
```

The key is our function is estimation-method agnostic: we will use it for each of our three estimators. Here we evaluate our 'mean' estimator:

```
filter( raw.exps, estimator == "mean" ) %>%
    pull( estimate ) %>%
    estimator.quality( mu = mu )
```

```
##       RMSE        bias        SE
## 1 0.318817 -0.01070817 0.3187965
```

Aside: Perhaps, code-wise, the above is piping having gone too far? If you don't like this style, you can do this:

```
estimator.quality( raw.exps$estimate[ raw.exps$estimator=="mean"], mu )
```

```
##       RMSE        bias        SE
## 1 0.318817 -0.01070817 0.3187965
```

To do all our three estimators, we group by estimator and evaluate for each estimator. In tidyverse 1.0 `summarise` can handle multiple responses, but they will look a bit weird in our output, hence the 'unpack()' argument which makes each column its own column (if we do not unpack, we have a "data frame column" which is an odd thing).

```
raw.exps %>%
    group_by( estimator ) %>%
    summarise( qual = estimator.quality( estimate, mu = 0 ) ) %>%
    tidyr::unpack( cols=c(qual) )
```

```
## # A tibble: 3 x 4
##   estimator  RMSE     bias    SE
##   <chr>     <dbl>    <dbl> <dbl>
## 1 mean      0.319 -0.0107  0.319
## 2 median    0.316 -0.00708 0.316
## 3 trim.mean 0.292 -0.00963 0.292
```

We then pack up the above into a function, as usual. Our function takes our two parameters of sample size and degrees of freedom, and returns a data frame of results.

```
run.simulation = function( n, df0 ) {
    raw.exps <- replicate( 1000, {
        dt = gen.data( n=n, df0=df0 )
        analyze.data( dt )
    }, simplify = FALSE )
    raw.exps = bind_rows( raw.exps, .id = "runID" )

    rs <- raw.exps %>%
        group_by( estimator ) %>%
        summarise( qual = estimator.quality( estimate, mu = 0 ) ) %>%
        tidyr::unpack( cols=c( qual ) )

    rs
}
```

Our function will take our two parameters, run a simulation, and give us the results. We see here that none of our estimators are particularly biased and the trimmed mean has, possibly, the smallest RMSE, although it is a close call.

```
run.simulation( 10, 5 )
```

```
## # A tibble: 3 x 4
##   estimator  RMSE    bias    SE
##   <chr>     <dbl>   <dbl> <dbl>
## 1 mean      0.306 0.00848 0.306
## 2 median    0.309 0.00121 0.309
## 3 trim.mean 0.283 0.00388 0.283
```

Ok, now we want to see how sample size impacts our different estimators. If we also vary degrees of freedom we have a *three*-factor experiment, where one of the factors is our estimator itself. We are going to use a new clever trick. As before, we use `pmap()`, but now we store the entire dataframe of results we get back from our function in a new column of our original dataframe. See R for DS, Chapter 25.3. This trick works best if we have everything as a `tibble` which is basically a dataframe that prints a lot nicer and doesn't try to second-guess what you are up to all the time.

```
ns = c( 10, 50, 250, 1250 )
dfs = c( 3, 5, 15, 30 )
lvls = expand_grid( n=ns, df=dfs )
```

```
# So it stores our dataframe results in our lvls data properly.
lvls = as_tibble(lvls)

results <- lvls %>% mutate( results = pmap( lvls, run.simulation ) )
```

We have stored our results (a bunch of dataframes) in our main matrix of simulation runs.

```
print( results, n=4 )
```

```
## # A tibble: 16 x 3
##       n    df results
##   <dbl> <dbl> <list>
## 1    10     3 <tibble [3 x 4]>
## 2    10     5 <tibble [3 x 4]>
## 3    10    15 <tibble [3 x 4]>
## 4    10    30 <tibble [3 x 4]>
## # ... with 12 more rows
```

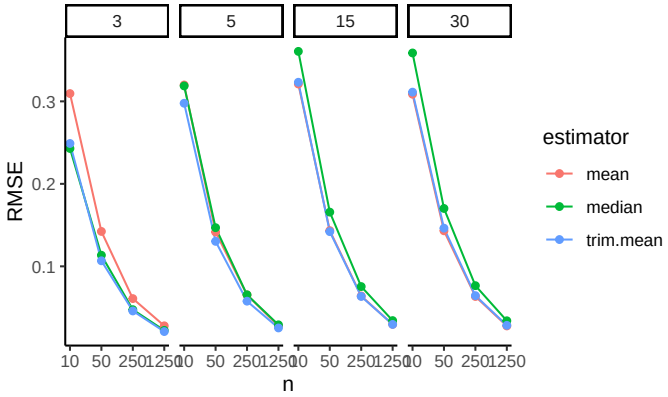
The `unnest()` function will stack up our dataframes, replicating the other columns in the main dataframe so it makes a nice rectangular dataset, all nice like. See (hard to read) R for DS Chapter 25.4.

```
results <- unnest( results, cols="results" )
results
```

```
## # A tibble: 48 x 6
##        n    df estimator  RMSE     bias    SE
##    <dbl> <dbl> <chr>     <dbl>    <dbl> <dbl>
##  1    10     3 mean      0.309 -0.0119  0.309
##  2    10     3 median    0.243 -0.0102  0.243
##  3    10     3 trim.mean 0.249 -0.0120  0.249
##  4    10     5 mean      0.320  0.00627 0.320
##  5    10     5 median    0.319  0.0121  0.319
##  6    10     5 trim.mean 0.298  0.00692 0.298
##  7    10    15 mean      0.321  0.00444 0.321
##  8    10    15 median    0.361  0.00925 0.361
##  9    10    15 trim.mean 0.323  0.00801 0.323
## 10    10    30 mean      0.309 -0.00229 0.309
## # ... with 38 more rows
```

And plot:

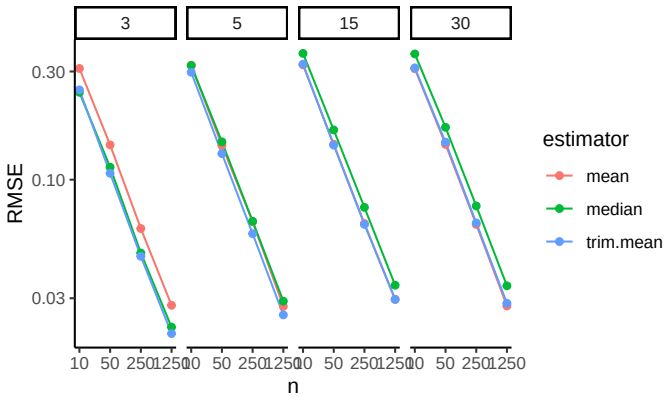
```
ggplot( results, aes(x=n, y=RMSE, col=estimator) ) +
            facet_wrap( ~ df, nrow=1 ) +
            geom_line() + geom_point() +
            scale_x_log10( breaks=ns )
```



14.5 Improving the visualization of the results

The above doesn't show differences clearly because all the RMSE goes to zero. It helps to log our outcome, or otherwise rescale. The logging version shows differences are relatively constant given changing sample size.

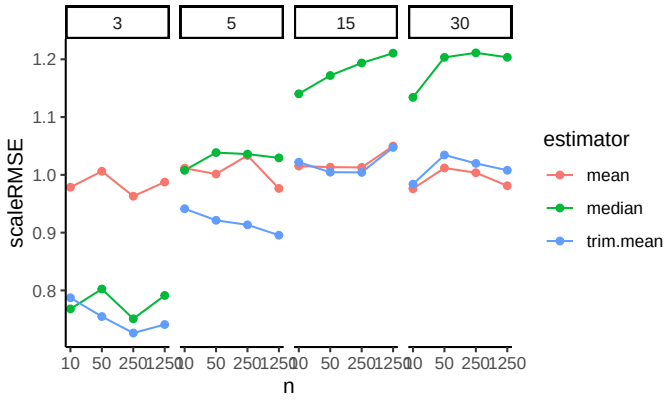
```
ggplot( results, aes(x=n, y=RMSE, col=estimator) ) +
    facet_wrap( ~ df, nrow=1 ) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns ) +
    scale_y_log10()
```



Better is to rescale using our knowledge of standard errors. If we scale by the square root of sample size, we should get horizontal lines. We now clearly see the trends.

```
results <- mutate( results, scaleRMSE = RMSE * sqrt(n) )
```

```
ggplot( results, aes(x=n, y=scaleRMSE, col=estimator) ) +
    facet_wrap( ~ df, nrow=1) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns )
```



Overall, we see the scaled error of the mean it is stable across the different distributions. The trimmed mean is a real advantage when the degrees of freedom are small. We are cropping outliers that destabilize our estimate which leads to great wins. As the distribution grows more normal, this is no longer an advantage and we get closer to the mean in terms of performance. Here we are penalized slightly bye having dropped 10% of our data, so the standard errors will be slightly larger.

The median is not able to take advantage of the nuances of a data set because it is entirely determined by the middle value. When outliers cause real concern, this cost is minimal. When outliers are not a concern, the median is just worse.

Overall, the trimmed mean seems an excellent choice: in the presence of outliers it is far more stable than the mean, and when there are no outliers the cost of using it is small.

In terms of thinking about designing simulation studies, we see clear visual displays of simulation results can tell very clear stories. Eschew complicated tables with lots of numbers.

14.6 Extension: The Bias-variance tradeoff

We can use the above simulation to examine these same estimators when we the median is not the same as the mean. Say we want the mean of a distribution, but have systematic outliers. If we just use the median, or trimmed mean, we might have bias if the outliers tend to be on one side or another. For example, consider the exponential distribution:

```
nums = rexp( 100000 )
mean( nums )
```

```
## [1] 0.9990221
```

```
mean( nums, trim=0.1 )
```

```
## [1] 0.83185
```

```
median( nums )
```

```
## [1] 0.6940772
```

Our trimming, etc., is *biased* if we think of our goal as estimating the mean. But if the trimmed estimators are much more stable, we might still wish to use them. Let's find out.

Let's generate a mixture distribution, just for fun. It will have a nice normal base with some extreme outliers. We will make sure the overall mean, including the outliers, is always 1, however. (So our target, μ is now 1, not 0.)

```
gen.data.exp = function( n, prob.outlier = 0.05 ) {
    nN = rbinom( 1, n, prob.outlier )
    nrm = rnorm( n - nN, mean=0.5, sd=1 )
    outmean = (1 - (1-prob.outlier)/2) / prob.outlier
    outs = rnorm( nN, mean=outmean, sd=10 )
    c( nrm, outs )
}
```

Let's look at our distribution

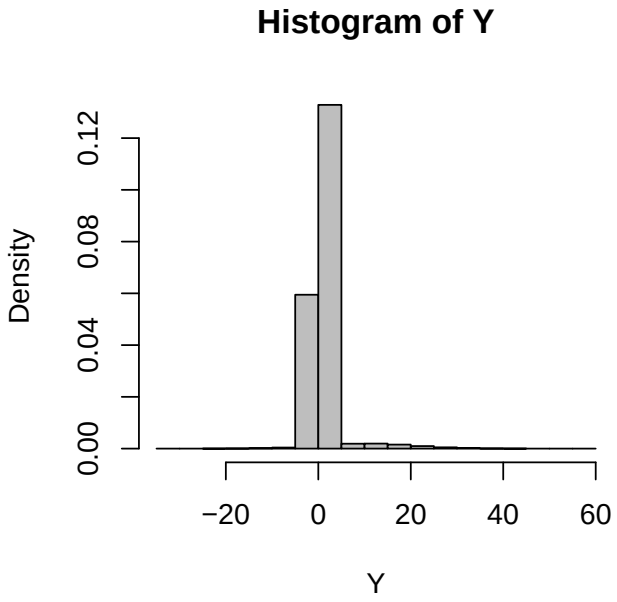
```
Y = gen.data.exp( 10000000, prob.outlier = 0.05 )
mean( Y )
```

```
## [1] 1.000219
```

```
sd( Y )
```

```
## [1] 3.271463
```

```
hist( Y, breaks=30, col="grey", prob=TRUE )
```



We steal the code from above, modifying it slightly for our new function and changing our target parameter from 0 to 1:

```
run.simulation.exp = function( n ) {
    raw.exps <- replicate( 1000, {
        dt = gen.data.exp( n=n )
        analyze.data( dt )
    }, simplify = FALSE )
    raw.exps = bind_rows( raw.exps, .id = "runID" )

    rs <- raw.exps %>%
        group_by( estimator ) %>%
        summarise( qual = estimator.quality( estimate, mu = 1 ) ) %>%
        tidyr::unpack( cols = c( qual ) )
```

```
    rs
}

res = run.simulation.exp( 100 )
res
```

```
## # A tibble: 3 x 4
##   estimator  RMSE     bias    SE
##   <chr>     <dbl>    <dbl> <dbl>
## 1 mean      0.315  0.00185 0.315
## 2 median    0.467 -0.450   0.127
## 3 trim.mean 0.450 -0.436   0.110
```

And for our experiment we vary the sample size

```
ns = c( 10, 20, 40, 80, 160, 320 )
lvls = tibble( n=ns )
```

```
results <- lvls %>%
  mutate( results = pmap( lvls, run.simulation.exp ) ) %>%
  unnest( cols = c(results) )
head( results )
```

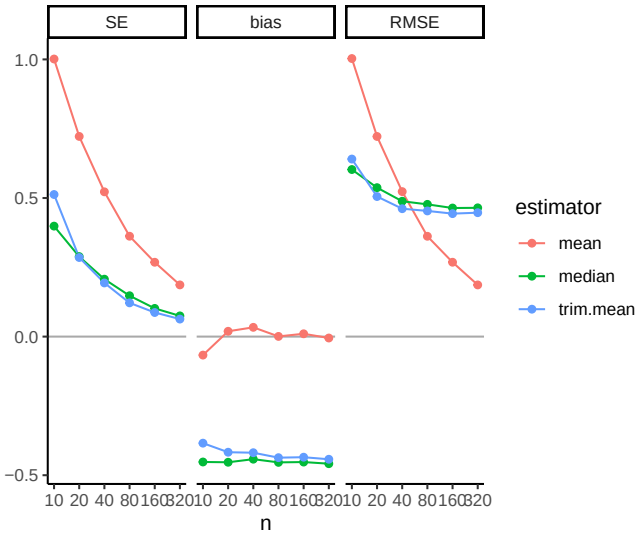
```
## # A tibble: 6 x 5
##       n estimator  RMSE    bias    SE
##   <dbl> <chr>     <dbl>   <dbl> <dbl>
## 1    10 mean      1.00  -0.0667 1.00
## 2    10 median    0.603 -0.452  0.399
## 3    10 trim.mean 0.641 -0.384  0.513
## 4    20 mean      0.722  0.0192 0.722
## 5    20 median    0.537 -0.453  0.289
## 6    20 trim.mean 0.505 -0.417  0.285
```

Here we are going to plug multiple outcomes. Often with the simulation study we are interested in different measures of performance. For us, we want to know the standard error, bias, and overall error (RMSE). To plot this we first gather our outcomes to make a long form dataframe of results:

```
res2 = gather( results, RMSE, bias, SE, key="Measure",value="value" )
res2 = mutate( res2, Measure = factor( Measure, levels=c("SE","bias","RMSE" )))
```

And then we plot, making a facet for each outcome of interest:

```
ggplot( res2, aes(x=n, y=value, col=estimator) ) +
    facet_grid( . ~ Measure ) +
    geom_hline( yintercept=0, col="darkgrey" ) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns ) +
    labs( y="" )
```



We see how different estimators have different biases and different uncertainties. The bias is negative for our trimmed estimators because we are losing the big outliers above and so getting answers that are too low.

The RMSE captures the trade-off in terms of what estimator gives the lowest overall *error*. For this distribution, the mean wins as the sample size increases because the bias basically stays the same and the SE drops. But for smaller samples the trimming is superior. The median (essentially trimming 50% above and below) is overkill and has too much negative bias.

From a simulation study point of view, notice how we are looking at three different qualities of our estimators. Some people really care about bias, some care about RMSE. By presenting all results we are transparent about how the different estimators operate.

Next steps would be to also examine the associated estimated standard errors for the estimators, seeing if these estimates of estimator uncertainty are good or poor. This leads to investigation of coverage rates and similar.

Chapter 15

Parallel Processing

Especially if you take our advice of "when in doubt, go more general" and if you calculate monte carlo standard errors, you will quickly come up against the limits of your computer. Simultions can be incredibly computationally intensive, and there are a few means for dealing with that. The first, touched on at times throughout the book, is to optimize ones code by looking for ways to remove extraneous calculation (e.g., by writing ones own methods rather than using the safety-checking and thus sometimes slower methods in R, or by saving calculations that are shared across different estimation approaches). The second is to use more computing power. This latter approach is the topic of this chapter.

There are two general ways to do parallel calculation. The first is to take advantage of the fact that most modern computers have multiple cores (i.e., computers) built in. With this approach, we tell R to use more of the processing power of your desktop or laptop. If your computer has eight cores, you can easily get a near eight-fold increase in the speed of your simulation.

The second is to use cloud computing, or compute on a cluster. A computing cluster is a network of hundreds or thousands of computers, coupled with commands where you break apart a simulation into pieces and send the pieces to your army of computers. Conceptually, this is the same as when you do baby parallel on your desktop: more cores equals more simulations per minute and thus faster simulation overall. But the interface to a cluster can be a bit tricky, and very cluster-dependent.

But once you get it up and running, it can be a very powerful tool. First, it takes the computing off your computer entirely, making it easier to set up a job to run for days or weeks without making your day to day life any more difficult. Second, it gives you hundreds of cores, potentially, which means a speed-up of hundreds rather than four or eight.

Simulations are a very natural choice for parallel computation. With a multifactor experiment it is very easy to break apart the overall into pieces. For

example, you might send each factor combination to a single machine. Even without multi factor experiments, due to the cycle of "generate data, then analyze," it is easy to have a bunch of computers doing the same thing, with a final collection step where all the individual iterations are combined into one at the end.

15.1 Parallel on your computer

Most modern computers have multiple cores, so you can run a parallel simulation right in the privacy of your own home!

To assess how many cores you have on your computer, you can use the `detectCores()` method in the `parallel` package:

```
parallel::detectCores()
```

```
## [1] 12
```

Normally, unless you tell it to do otherwise, ***R only uses one core***. This is obviously a bit lazy on R's part. But it is easy to take advantage of multiple cores using the `future` and `furrr` packages.

```
library(future)
library(furrr)
```

In particular, the `furrr` package replicates our `map` functions, but in parallel. We first tell our R session what kind of parallel processing we want using the `future` package. In general, using `plan(multisession)` is the cleanest: it will start one entire R session per core, and have each session do work for you. The alternative, `multicore` does not seem to work well with Windows machines, nor with RStudio in general.

The call is simple:

```
plan(multisession, workers = parallel::detectCores() - 1 )
```

The `workers` parameter specifies how many of your cores you want to use. Using all but one will let your computer still operate mostly normally for checking email and so forth. You are carving out a bit of space for your own adventures.

Once you set up your plan, you use `future_pmap()`; it works just like `pmap()` but evaluates across all available workers specified in the plan call. Here we are running a parallel version of the multifactor experiment discussed in Chapter @ref(exp_design) (see chapter @ref(case_Cronback) for the simulation itself).

```
tictoc::tic()
params$res = future_pmap(params,
                         .f = run_alpha_sim,
                         .options = furrr_options(seed = NULL))
tictoc::tic()
```

Note the `.options = furrr_options(seed = NULL)` part of the argument. This is to silence some warnings. Given how tasks are handed out, R will get upset if you don't do some handholding regarding how it should set seeds for pseudoranom number generation. In particular, if you don't set the seed, the multiple sessions could end up having the same starting seed and thus run the exact same simulations (in principle). We have seen before how to set specific seed for each simulation scenario, but `furrr` doesn't know we have done this. This is why the extra argument about seeds: it is being explicit that we are handling seed setting on our own.

We can compare the running time to running in serial (i.e. using only one worker):

```
tictoc::tic()
params$res2 = dplyr::select(params, n:seed) %>%
  pmap(.f = run_alpha_sim)
tictoc::tic()
```

(The `select` command is to drop the `res` column from the parallel run; it would otherwise be passed as as parameter to `run_alpha_sim` which would in turn cause an error due to the unrecognized parameter.)

15.2 Parallel off your computer

In general, a "cluster" is a system of computers that are connected up to form a large distributed network that many different people can use to do large computational tasks (e.g., simulations!). These clusters will have some overlaying coordinating programs that you, the user, will interact with to set up a "job," or set of jobs, which is a set of tasks you want some number of the computers on the cluster to do for you in tandum.

These coordinating programs will differ, depending on what cluster you are using, but have some similarities that bear mention. For running simulations, you only need the smallest amount of knowledge for engaging with these systems, which is a good thing, because you don't really need all the individual computers communicating with each other at all (which is the hard part, in general).

15.2.1 What is a command-line interface?

In the good ol' days, when things were simpler, yet more difficult, you would interact with your computer via a "command-line interface." The easiest way to think about this is as an R console, but in a different language that the entire computer speaks. It is designed to do things like find files with a specific name, or copy entire directories, or, importantly, start different programs. Another place you may have used this inteface is when working with Git: anything fancy with Git is often done via command-line. People will talk about a "shell" (a generic term for this computer interface) or "bash" or "csh." You can get access to a shell from within RStudio by clicking on the "Terminal" tab. Try it, if you've never done anything like this before, and type

```
ls
```

It should list some files. Note this command does *not* have the parenthesis after the command, like in R or a programming language. The syntax of a shell is usually mystifying and brutal: it is best to just steal scripts from the internet and try not to think about it too much, unless you want to think about it a lot.

In particular, from the command line interface you can start an R program, telling it to start up and run a script for you. This way of running R is non-interactive: you say "go do this thing," and R starts up, goes and does it, and then quits. Any output R generates on the way will be saved in a file, and any files you save along the way will also be at your disposal once R has completed.

To see this in action make the following script in a file called "dumb_job.R":

```
library( tidyverse )
cat( "Making numbers\n" )
Sys.sleep(30)
cat( "Now I'm ready\n" )
dat = tibble( A = rnorm( 1000 ), B = runif( 1000 ) * A )
write_csv( dat, file="sim_results.csv" )
Sys.sleep(30)
cat( "Finished\n" )
```

Then open the terminal and run:

```
R CMD BATCH dumb_job.R R_output.txt --no-save
```

The above command says "Run R" (the first part) in batch mode (the "CMD BATCH" part) where you source the `dumb_job.R` script, saving all console output in the file `R_output.txt` (it will be saved in the current directory where you run the program), and where you don't save the workspace when finished.

This command should take about a minute to complete. Once it is finished, verify that you have the `R_output.txt` and the data file `sim_results.csv`. You can actually see `R_output.txt` half-way through, as it is running: you will see the usual header of R telling you what it loading, and all of that. It is saving everything as you go.

Running R in this fashion is the key element to a basic way of setting up a massive job on the cluster: you will have a bunch of R programs all "going and doing something" on different computers in the cluster. They will all save their results to files (they will have files of different names, or you will not be happy with the end result) and then you will gather these files together to get your final set of results.

Small Exercise: Try putting an error in your `dumb_job.R` script. What happens when you run it in batch mode?

15.2.2 Running a job on a cluster

In the above, you can run a command on the command-line, and it will pause while it runs. Note how, when you hit return, the program just sits there for a minute before you get your command-line prompt back.

When you run a program on a cluster, it doesn't quite work that way. You instead set a program to run, but it runs somewhere else. You get your command-line prompt back, and can do other things, while the program runs in the background.

There are various methods for doing this, but they usually boil down to a request from you to some sort of managerial process that takes requests and assigns some computer, somewhere, to do them. (Imagine a dispatcher at a taxi company. You call up, ask for a ride, and it sends you a taxi to do it. The dispatcher is just fielding requests, assinging them to taxis.)

For example, one dispatcher is the slurm (which may or may not be on the cluster you are attempting to use; this is where a lot of this information gets very cluster-specific).

You first set up a script that describes the job to be run. It is like a work request. This would be a plain text file, such as this example (`sbatch_runScript.txt`):

```
#!/bin/bash
#SBATCH -n 32                                                  # Number of cores requested
#SBATCH -N 1                                                     # Ensure that all cores are on
#SBATCH -t 480                                                 # Runtime in minutes
#SBATCH -p stats                                               # Partition to submit to
#SBATCH --mem-per-cpu=1000                    # Memory per cpu in MB
#SBATCH --open-mode=append                          # Append to output file, don't truncate
#SBATCH -o /output/directory/out/%j.out # Standard out goes to this file
```

```
#SBATCH -e /output/directory/out/%j.err # Standard err goes to this file
#SBATCH --mail-type=ALL                          # Type of email notification- BEGIN,ENI
#SBATCH --mail-user=email@gmail.com         # Email address

# You might have some special loading of modules in the computing environment
source new-modules.sh
module load gcc/7.1.0-fasrc01
module load R
export R_LIBS_USER=$HOME/apps/R:$R_LIBS_USER

#R file to run, and txt files to produce for output and errors
R CMD BATCH estimator_performance_simulation.R logs/R_output_${INDEX_VAR}.txt --no-save
```

This file starts with a bunch of variables that describe how sbatch should handle the request. It then has a series of commands that get the computer environment ready. Finally, it has the `R CMD BATCH` command that does the work you want.

These scripts can be quite confusing to understand. There are so many options! What do these things even do? The answer is, for researchers early on their journey to do this kind of work, "Who knows?" The general rule is to find an example file for the system you are working on that works, and then modify it for your own purposes.

Once you have sucha file, you could run it on the command line, like this:

```
sbatch -o stdout.txt \
        --job-name=my_script \
        sbatch_runScript.txt
```

You do this, and it will *not* sit there and wait for the job to be done. The `sbatch` command will instead send the job off to some computer which will do the work in parallel.

Interestingly, your R script could, at this point, do the "one computer" parallel type code listed above. Note the script above has 32 cores; your single job could then have 32 cores all working away on their individual pieces of the simulation, as before (e.g., with `future_pmap`). You would have a 32-fold speedup, in this case.

This is the core element to having your simulation run on a cluster. The next step is to do this *a lot*, sending off a bunch of these jobs to different computers.

Some final tips

- Remember to save a workspace or RDS!! Once you tell Odyssey to run an R file, it, well, runs the R file. But, you probably want information after it's done - like an R object or even an R workspace. For any R file

you want to run on Odyssey, remember at the end of the R file to put a command to save something after everything else is done. If you want to save a bunch of R objects, an R workspace might be a good way to go, but those files can be huge. A lot of times I find myself wanting only one or two R objects, and RDS files are a lot smaller.

- Moving files from a cluster to your computer. You will need to first upload your files and code to the cluster, and then, once you've saved your workspace/RDS, you need those back on your computer. Using a scp client such as FileZilla is an easy way to do this file-transfer stuff. You can also use a Git repo for the code, but checking in the simulation results is not generally advisable: they are big, and not really in the spirit of a verson control system. Download your simulation results outside of Git, and keep your code in Git, is a good rule of thumb.

15.2.3 Checking on a job

Once your job is working on the cluster, it will keep at it until it finishes (or crashes, or is terminated for taking up too much memory or time). As it chugs away, there will be different ways to check on it. For example, you can, from the console, list the jobs you have running to see what is happening:

```
sacct -u lmiratrix
```

except, of course, "`lmiratrix`" would be changed to whatever your username is. This will list if your file is running, pending, timed out, etc. If it's pending, that usually means that someone else is hogging up space on the cluster and your job request is in a queue waiting to be assigned.

The `sacct` command is customizable, e.g.,

```
sacct -u lmiratrix --format=JobID,JobName%30,State
```

will not truncate your job names, so you can find them more easily.

You can check on a specific job, if you know the ID:

```
squeue -j JOBID
```

Something that's fun is you can check who's running files on the stats server by typing:

```
showq-slurm -p stats -o
```

You can also look at the log files

```
tail my_log_file.log
```

to see if it is logging information as it is working.

The email arguments, above, cause the system to email you before and after the job is complete. The email notifications you can choose are `BEGIN`, `END`, `FAIL`, and `ALL`; `ALL` is generally good. What is a few more emails?

15.2.4 Running lots of jobs on a cluster

The above shows how to fire off a job (possibly a big job) that can run over days or weeks to give you your results. There is one more piece that can allow you to use even more computing resources to do things even faster, which is to do a whole bunch of job requests like the above, all at once. This multiple dispatching of sbatch commands is the final piece one might need for large simulations on a cluster: you are setting in motion a bunch of processes, each set to a specific task.

Asking for multiple, smaller, jobs is also nicer for the cluster than having one giant job that goes on for a long time. By dividing a job into smaller pieces, and asking the scheduler to schedule those pieces, you can let the scheduler share and allocate resources between you and others more fairly. It can make a list of your jobs, and farm them out as it has space. This might go faster for you; with a really big job, the scheduler can't even allocate it until the needed number of workers is available. With smaller jobs, you can take a lot of little spaces to get your work done. Especially since simulation is so independent (just doing the same thing over and over) there is no need for one giant thing.

To make multiple, related, requests, we create a for-loop in the Terminal to make a whole series sbatch requests. Then, each sbatch request will do one part of the overall simulation. We can write this program in the shell, just like you can write R scripts in R. A shell scripts does a bunch of shell commands for you, and can even have variables and loops and all of that fun stuff.

For example, the following `run_full_simulation.sh` is a script that fires off a bunch of jobs for a simulation. Note that it makes a variable `INDEX_VAR`, and sets up a loop so it can run 500 tasks indexed 1 through 500.

The first `export` line adds a collection of R libraries to the path (the list of places, called `R_LIBS_USER` where R will look for libraries). It also makes log files and names each job with the index so you can know who is generating what file.

```
export R_LIBS_USER=$HOME/apps/R:$R_LIBS_USER

for INDEX_VAR in $(seq 1 500); do
```

```
  #print out indexes
  echo "${INDEX_VAR}"

  #give indexes to R so it can find them.
  export INDEX_VAR

  #Run R script, and produce output files
  sbatch -o logs/sbout_p${INDEX_VAR}.stdout.txt \
        --job-name=runScr_p${INDEX_VAR} \
        sbatch_runScript.txt

  sleep 1 # pause to be kind to the scheduler

done
```

One question is then how do the different processes know what part of the simulation they should be working on? E.g., they each need to have their own seed so they don't do exactly the same simulation. They need their own filenames so they save things in their own files. The key is the `export INDEX_VAR` line: this puts a variable in the environment that will be set to a specific number. Inside your R script, you can get that index like so:

```
index <- as.numeric(as.character(Sys.getenv("INDEX_VAR")))
```

You can then use the index to make unique filenames when you save your results, so each process has its own filename:

```
filename = paste0( "raw_results/simulation_results_", index, _".rds" )
```

You can also modify your seed such as with:

```
factors = mutate( factors,
                  seed = set.seed( 1000 * seed + index ) )
```

Now even if you have a series of seeds within the simulation script (as we have seen before), each script will have unique seeds not shared by any other script (assuming you have fewer than 1000 separate job requests).

15.2.5 Resources for Harvard's Odyssey

The above guidiance is tailored for Harvard' computing environment, primarily. For that environment in particular, there are many additional resources such as:

- Odyssey Guide: https://rc.fas.harvard.edu/resources/odyssey-quickstart-guide/
- R on Odyssey: https://rc.fas.harvard.edu/resources/documentation/software/r/

For installing R packages so they are seen by the scripts run by sbatch, see (https://www.rc.fas.harvard.edu/resources/documentation/software-on-odyssey/r/)

Other clusters should have similar documents giving needed guidance for their specific contexts.

15.2.6 Acknowledgements

Some of the above material is based on tutorials built by Kristen Hunter and Zach Branson, past doctoral students of Harvard's statistics department.

Chapter 16

Case study: The Power and Validity of Neyman's ATE Estimate

The way to build a simulation experiment is to first write code to run a specific simulation for a specific scenario. Once that is working, we will re-use the code to systematically explore a variety of scenarios so we can see how things change as scenario changes. Next I would build up this system.

For our running example we are going to look at a randomized experiment. We will assume the treatment and control groups are normally distributed with two different means. We will generate a random data set, estimate the treatment effect by taking the difference in means and calculating the associated standard error, and generating a p-value using the normal approximation. (As we will see, this is not a good idea for small sample size since we should be using a t-test style approach.)

16.1 Step 1: Write a function for a specific simulation given specific parameters.

Our function will generate two groups of the given sizes, one treatment and one control, and then calculate the difference in means. It will then test this difference using the normal approximation.

The function also calculates and returns the effect size as the treatment effect divided by the control standard deviation (useful for understanding power, shown later on).

```
run.one = function( nC, nT, sd, tau, mu = 5 ) {
  Y0 = mu + rnorm( nC, sd=sd )
  Y1 = mu + tau + rnorm( nT, sd=sd )

  tau.hat = mean( Y1 ) - mean( Y0 )
  SE.hat = sqrt( var( Y0 ) / ( nC ) + var( Y1 ) / ( nT ) )

  z = tau.hat / SE.hat
  pv = 2 * (1 - pnorm( abs( z ) ))

  data.frame( tau.hat = tau.hat, ES = tau / sd, SE.hat = SE.hat, z=z, p.value=pv )
}
```

A single run will generate a data set, analyze it, and give us back a variety of results as a one-row dataframe. It will turn out that having results in a dataframe makes using the tidyverse easier.

```
run.one( nT=5, nC=10, sd=1, tau=0.5 )
```

```
##     tau.hat  ES    SE.hat        z   p.value
## 1 0.0540692 0.5 0.5395857 0.100205 0.9201816
```

16.1.1 Running our single trial more than once

We `replicate()` and then stack everything to get all our results.

```
eres <- replicate( 500, run.one( nC=10, nT=10, sd=1, tau=0.5 ), simplify = FALSE )
eres = bind_rows( eres )
```

Now we see why the dataframe return value of `run.one` is useful: we can then stack all of our simulation runs to get our final set of results. Each row is a simulation run: head(eres)

We then summarize our results with the `dplyr summarise` function. Our summarization calculates the average treatment effect estimate `E.tau.hat`, the average Standard Error estimate `E.SE.hat`, the average Effect Size `ES`, and the power `power` (defined as the percent of time we reject at alpha=0.05, i.e., the percent of times our p-value was less than our 0.05 threshold):

```
eres %>% summarise( E.tau.hat = mean( tau.hat ),
                    E.SE.hat = mean( SE.hat ),
                    ES = mean( ES ),
                    power = mean( p.value <= 0.05 ) )
```

```
##   E.tau.hat  E.SE.hat  ES power
## 1 0.5308502 0.4343176 0.5 0.242
```

We bundle the above into a function that runs our single trial multiple times and summarizes the results:

```
run.experiment = function( nC, nT, sd, tau, mu = 5, R = 500 ) {

  eres = replicate( R, run.one( nC, nT, sd, tau, mu ), simplify=FALSE )
  eres = bind_rows( eres )
  eres %>% summarise( E.tau.hat = mean( tau.hat ),
                      E.SE.hat = mean( SE.hat ),
                      ES = mean( ES ),
                      power = mean( p.value <= 0.05 ) ) %>%
    mutate( nC=nC, nT=nT, sd=sd, tau=tau, mu=mu, R=R )
}
```

Our function also adds in the details of the simulation (the parameters we passed to the `run.one()` call). This is an easy way to keep track of things.

Test our function to see what we get:

```
run.experiment( 10, 3, 1, 0.5 )
```

```
##   E.tau.hat  E.SE.hat  ES power nC nT sd tau mu   R
## 1 0.5032869 0.6139875 0.5 0.208 10  3  1 0.5  5 500
```

Key point: We also want a dataframe back from `run.experiment()`, because, after calling `run.experiment()` many times, we are going to stack the results up to make one long dataframe of results. Happily the `dplyr` package gives us dataframes so this is not a problem here.

16.2 Step 2: Make a dataframe of all experimental combinations desired

We use the above to run a *multi-factor simulation experiment*. We are going to vary four factors: control group size, treatment group size, standard deviation of the units, and the treatment effect.

We first set up the levels we want to have for each of our factors (these are our *simulation parameters*).

```
nC = c( 2, 4, 7, 10, 50, 500 )
nT = c( 2, 4, 7, 10, 50, 500 )
sds = c( 1, 2 )
tau = c( 0, 0.5, 1 )
```

We then, using `expand_grid()` generate a dataframe of all combinations of our factors.

```
experiments = expand_grid( nC=nC, nT=nT, sd=sds, tau=tau )
experiments
```

```
## # A tibble: 216 x 4
##       nC    nT    sd   tau
##    <dbl> <dbl> <dbl> <dbl>
##  1     2     2     1   0
##  2     2     2     1   0.5
##  3     2     2     1   1
##  4     2     2     2   0
##  5     2     2     2   0.5
##  6     2     2     2   1
##  7     2     4     1   0
##  8     2     4     1   0.5
##  9     2     4     1   1
## 10     2     4     2   0
## # ... with 206 more rows
```

See what we get? One row will correspond to a single experimental run. Note how the parameters we would pass to `run.experiment()` correspond to the columns of our dataset.

Also, is easy to end up running a lot of experiments! In this case we have 216!

We next run an experiment for each row of our dataframe of experiment factor combinations using the `pmap_df()` function which will, for each row in our dataframe, call `run.experiment()`, passing one parameter taken from each column of our dataframe.

```
exp.res <- experiments %>% pmap_df( run.experiment, R=500 )
```

The `R=500` after `run.experiment` passes the *same* parameter of $R = 500$ to each run (we run the same number of trials for each experiment).

Here is a peek at our results:

```
head( exp.res )
```

```
##    E.tau.hat  E.SE.hat   ES power nC nT sd tau mu   R
## 1 0.01311053 0.8724541 0.00 0.196  2  2  1 0.0  5 500
## 2 0.53992204 0.8640737 0.50 0.228  2  2  1 0.5  5 500
## 3 0.98325407 0.8929156 1.00 0.340  2  2  1 1.0  5 500
## 4 0.01209447 1.7077286 0.00 0.170  2  2  2 0.0  5 500
## 5 0.39500948 1.8184304 0.25 0.176  2  2  2 0.5  5 500
## 6 1.05321453 1.8179832 0.50 0.230  2  2  2 1.0  5 500
```

At this point you should save your simulation results to a file. This is especially true if the simulation happens to be quite time-intensive to run. Usually a csv file is sufficient.

We save using the tidyverse writing command; see "R for Data Science" textbook, 11.5.

```
dir.create("results", showWarnings = FALSE )
write_csv( exp.res, "results/simulation_results.csv" )
```

16.3 Step 3: Explore results

Once your simulation is run, you want to evaluate the results. One would often put this code into a seperate '.R' file that loads this saved file to start. This allows for easily changing how one analyzes an experiment without re-running the entire thing.

16.3.1 Visualizing experimental results

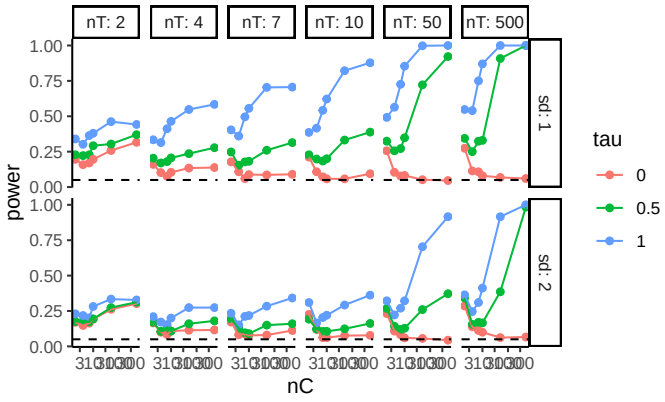
Plotting is always a good way to vizualize simulation results. Here we make our tau and ES into factors, so `ggplot` behaves, and then plot all our experiments as two rows based on one factor (`sd`) with the columns being another (`nT`). (This style of plotting a bunch of small plots is called "many multiples" and is beloved by Tufte.) Within each plot we have the x-axis for one factor (`nC`) and multiple lines for the final factor (`tau`). The y-axis is our outcome of interest, power. We add a 0.05 line to show when we are rejecting at rates above our nominal α. This plot shows the relationship of 5 variables.

```
exp.res = read_csv( "results/simulation_results.csv" )
```

```
## Rows: 216 Columns: 10
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (10): E.tau.hat, E.SE.hat, ES, power, nC, nT, sd, tau, mu, R
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
exp.res = exp.res %>% mutate( tau = as.factor( tau ),
                              ES = as.factor( ES ) )
ggplot( exp.res, aes( x=nC, y=power, group=tau, col=tau ) ) +
  facet_grid( sd ~ nT, labeller=label_both ) +
  geom_point() + geom_line() +
  scale_x_log10() +
  geom_hline( yintercept=0.05, col="black", lty=2)
```



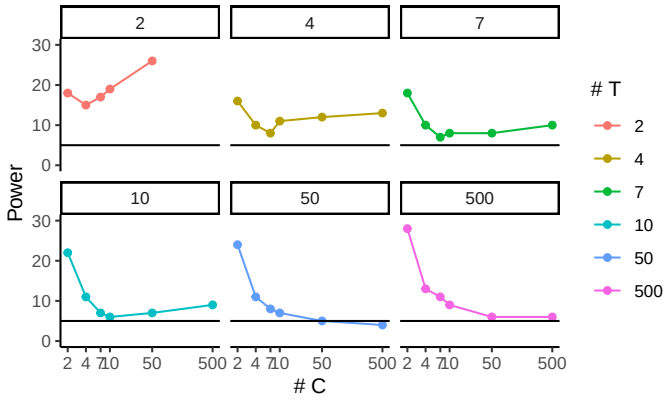
Note: We are seeing elevated rejection rates under the null for small and even moderate sample size!

We can zoom in on specific simulations run, to get some more detail such as estimated power under the null for larger groups. Here we check and we are seeing rejection rates of around 0.05, which is what we want. filter(exp.res, tau==0, nT >= 50, nC >= 50)

We can get fancy and look at rejection rate (power under `tau = 0`) as a function of both nC and nT using an interaction-style plot:

```
exp.res.rej <- exp.res %>% filter( tau == 0 ) %>%
  group_by( nC, nT ) %>%
  summarize( power = mean( power ) )

exp.res.rej = mutate( exp.res.rej, power = round( power * 100 ) )
```

```
ggplot( exp.res.rej, aes( x=nC, y=power, group=nT, col=as.factor(nT) ) ) +
  facet_wrap( ~ nT ) +
  geom_point() + geom_line() +
  geom_hline( yintercept = 5 ) +
  scale_y_continuous( limits = c( 0, 30 ) ) +
  scale_x_log10( breaks = unique( exp.res.rej$nC ) ) +
  labs( x = "# C", y = "Power", colour = "# T" )
```



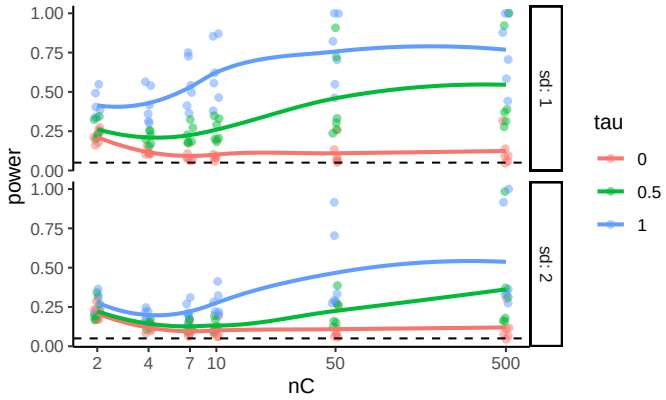
Note: This plot is somewhat imperfect. Increasing the number of simulation runs per trial could help smooth out the trends. That being said, we see that small tx or co groups are both bad here.

16.3.2 Looking at main effects

We can ignore a factor and just look at another. This is looking at the **main effect** or **marginal effect** of the factor.

The easy way to do this is to let `ggplot` smooth our individual points on a plot. Be sure to also plot the individual points to see variation, however.

```
ggplot( exp.res, aes( x=nC, y=power, group=tau, col=tau ) ) +
  facet_grid( sd ~ ., labeller=label_both ) +
  geom_jitter( width=0.02, height=0, alpha=0.5 ) +
  geom_smooth( se = FALSE ) +
  scale_x_log10( breaks=nC) +
  geom_hline( yintercept=0.05, col="black", lty=2)
```



Note how we see our individual runs that we marginalize over.

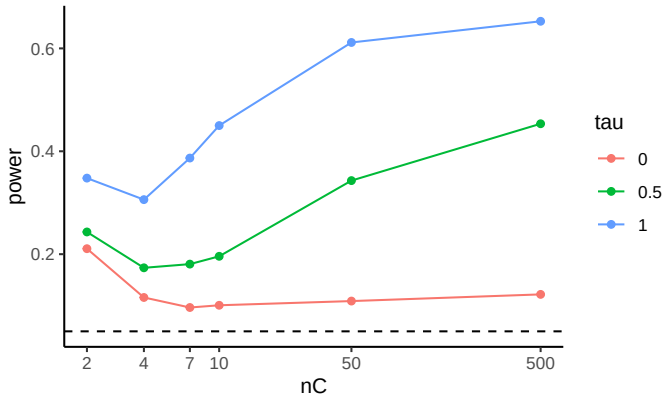
To look at our main effects we can also summarize our results, averaging our experimental runs across other factor levels. For example, in the code below we average over the different treatment group sizes and standard deviations, and plot the marginalized results.

To marginalize, we group by the things we want to keep. `summarise()` then averages over the things we want to get rid of.

```
exp.res.sum = exp.res %>% group_by( nC, tau ) %>%
  summarise( power = mean( power ) )
head( exp.res.sum )
```

```
## # A tibble: 6 x 3
## # Groups:   nC [2]
##      nC tau   power
##   <dbl> <fct> <dbl>
## 1     2 0     0.211
## 2     2 0.5   0.243
## 3     2 1     0.348
## 4     4 0     0.116
## 5     4 0.5   0.174
## 6     4 1     0.306
```

```
ggplot( exp.res.sum, aes( x=nC, y=power, group=tau, col=tau ) ) +
  geom_line() + geom_point() +
  scale_x_log10( breaks=nC) +
  geom_hline( yintercept=0.05, col="black", lty=2)
```

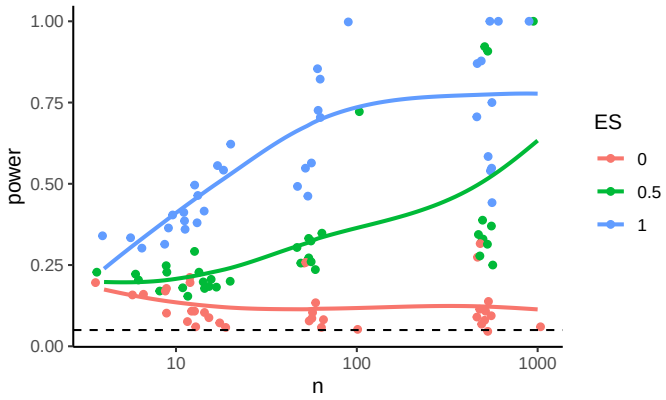


We can try to get clever and look at other aspects of our experimental runs. The above suggests that the smaller of the two groups is dictating things going awry, in terms of elevated rejection rates under the null. We can also look at things in terms of some other more easily interpretable parameter (here we switch to effect size instead of raw treatment effect).

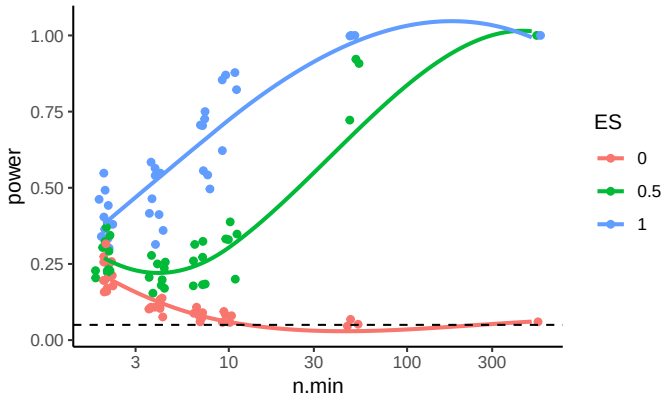
Given this, we might decide to look at total sample size or the smaller of the two groups sample size and make plots that way (we are also subsetting to just the `sd=1` cases as there is nothing really different about the two options; we probably should average across but this could reduce clarity of the presentation of results):

```
exp.res <- exp.res %>% mutate( n = nC + nT,
                               n.min = pmin( nC, nT ) )
```

```
ggplot( filter( exp.res, sd==1 ), aes( x=n, y=power, group=ES, col=ES ) ) +
  geom_jitter( width=0.05, height=0) +
  geom_smooth( se = FALSE, span = 1) +
  scale_x_log10() +
  geom_hline( yintercept=0.05, col="black", lty=2)
```




```
ggplot( filter( exp.res, sd==1 ), aes( x=n.min, y=power, group=ES, col=ES ) ) +
  geom_jitter( width=0.05, height=0) +
  geom_smooth( se = FALSE, span = 1) +
  scale_x_log10() +
  geom_hline( yintercept=0.05, col="black", lty=2)
```



Note the few observations out in the high `n.min` region for the second plot—this plot is a bit strange in that the different levels along the x-axis are assymetric with respect to each other. It is not balanced.

16.4 Addendum: Saving more details

Our `exp.res` dataframe from above has all our simulations, one simulation per row, with our measured outcomes. This is ideally all we need to analyze.

That being said, sometimes we might want to use a lot of disk space and keep much more. In particular, each row of `exp.res` corresponds to the summary of a whole collection of individual runs. We might instead store all of these runs.

To do this we just take the summarizing step out of our `run.experiment()`

```
run.experiment.raw = function( nC, nT, sd, tau, mu = 5, R = 500 ) {
  eres = replicate( R, run.one( nC, nT, sd, tau, mu ), simplify = FALSE )
  eres = bind_rows( eres )
  eres <- mutate( eres, nC=nC, nT=nT, sd=sd, tau=tau, mu=mu, R=R )
  eres
}
```

Each call to `run.experiment.raw()` gives one row per run. We replicate our simulation parameters for each row.

```
run.experiment.raw( 10, 3, 1, 0.5, R=4 )
```

```
##     tau.hat  ES    SE.hat         z     p.value nC nT sd tau mu R
## 1 0.6135525 0.5 0.8888169 0.6903025 0.490003969 10  3  1 0.5  5 4
## 2 0.2597831 0.5 0.5401115 0.4809804 0.630530436 10  3  1 0.5  5 4
## 3 0.4963473 0.5 0.5679538 0.8739219 0.382160761 10  3  1 0.5  5 4
## 4 1.3488370 0.5 0.4478926 3.0115187 0.002599444 10  3  1 0.5  5 4
```

The advantage of this is we can then generate new outcome measures, as they occur to us, later on. The disadvantage is this result file will be R times as many rows as the older file, which can get quite, quite large.

But disk space is cheap! Here we run the same experiment with our more complete storage. Note how the `pmap_df` stacks the multiple rows from each run, giving us everything nicely bundled up:

```
exp.res.full <- experiments %>% pmap_df( run.experiment.raw, R=500 )
head( exp.res.full )
```

```
##       tau.hat ES    SE.hat           z     p.value nC nT sd tau mu   R
## 1  0.05413216  0 1.2243659  0.04421240 0.964735098  2  2  1   0  5 500
## 2 -0.98301948  0 0.3004194 -3.27215676 0.001067304  2  2  1   0  5 500
## 3 -0.35344487  0 0.9861798 -0.35839802 0.720045476  2  2  1   0  5 500
## 4  0.34626891  0 1.3413763  0.25814450 0.796295390  2  2  1   0  5 500
## 5 -0.02718350  0 0.9248855 -0.02939121 0.976552583  2  2  1   0  5 500
## 6  1.36147713  0 0.8812838  1.54487938 0.122375441  2  2  1   0  5 500
```

We end up with a lot more rows:

```
nrow( exp.res.full )
```

```
## [1] 108000
```

```
nrow( exp.res )
```

```
## [1] 216
```

We next save our results:

```
write_csv( exp.res.full, "results/simulation_results_full.csv" )
```

Compare the file sizes: one is several k, the other is around 12 megabytes.

```
file.size("results/simulation_results.csv") / 1024
```

```
## [1] 13.47266
```

```
file.size("results/simulation_results_full.csv") / 1024
```

```
## [1] 10156.77
```

16.4.1 Getting results ready for analysis

If we generated raw results then we need to collapse them by experimental run before analyzing our results so we can explore the trends across the experiments. We do this by borrowing the summarise code from inside `run.experiment()`:

```
exp.res.sum <- exp.res.full %>%
  group_by( nC, nT, sd, tau, mu ) %>%
  summarise( R = n(),
             E.tau.hat = mean( tau.hat ),
             SE = sd( tau.hat ),
             E.SE.hat = mean( SE.hat ),
             ES = mean( ES ),
             power = mean( p.value <= 0.05 ) )
```

Note how I added an extra estimation of the true SE, just because I could! This is an easier fix, sometimes, than running all the simulations again after changing the `run.experiment()` method.

The results of summarizing during the simulation vs. after as we just did leads to the same place, however, although the order of rows in our final dataset are different (and we have a tibble instead of a data.frame, a consequence of using the `tidyverse`, but this is not something to worry about):

```
head( exp.res.sum )
```

```
## # A tibble: 6 x 11
## # Groups:   nC, nT, sd, tau [6]
##      nC    nT    sd   tau    mu     R E.tau.hat    SE E.SE.hat    ES power
##   <dbl> <dbl> <dbl> <dbl> <dbl> <int>     <dbl> <dbl>    <dbl> <dbl> <dbl>
## 1     2     2     1   0       5   500   -0.0137 0.925    0.902  0    0.152
## 2     2     2     1   0.5     5   500    0.550  1.00     0.881  0.5  0.208
## 3     2     2     1   1       5   500    0.982  0.996    0.879  1    0.318
## 4     2     2     2   0       5   500    0.0901 2.05     1.69   0    0.214
## 5     2     2     2   0.5     5   500    0.507  1.97     1.77   0.25 0.21
## 6     2     2     2   1       5   500    1.10   1.88     1.80   0.5  0.206
```

```
nrow( exp.res.sum )
```

```
## [1] 216
```

```
nrow( exp.res )
```

```
## [1] 216
```

Chapter 17

Design, analysis, and presentation of simulation results

You have all the pieces for writing simulations, but this still leaves open of what simulation should you run, and how do you present the results once you are finished.

17.1 Designing the simulation experiment

Recall the following design principles and acknowledgements:

- The primary limitation of simulation studies is **generalizability**.
- Choose conditions that allow you to relate findings to previous work.
- Err towards being comprehensive.
  - The goal should be to build an understanding of the major moving parts.
  - Presentation of results can always be tailored to illustrate trends.
- Explore breakdown points (e.g., what sample size is too small for applying a given method?).

17.2 Choosing parameter levels

You generally want to vary parameters that you believe matter, or that you think other people will believe matter. The first is so you can learn. The second

is to build your case.

Once you have identified your parameters you then have to decide on the levels of the parameter you will include in the simulation. There are three strategies you might take:

1. Vary a parameter over its entire range (or nearly so).
2. Choose parameter levels to represent realistic practical range.
    - Empirical justification based on systematic reviews of applications
    - Or at least informal impressions of what's realistic in practice
3. Choose parameters to emulate one important application.

In the above (1) is the most general—but also the most computationally intensive. (2) will focus attention, ideally, on what is of practical relevance to a practitioner. (3) is usually coupled with a subsequent applied data analysis, and in this case the simulation is often used to enrich that analysis. For example, if the simulation shows the methods work for data with the given form of the target application, people may be more willing to believe the application's findings.

Regardless of how you select your primary parameters, you should also vary nuisance parameters (at least a little) to test sensitivity of results. While simulations will (generally) never be fully generalizable, you can certainly make them so they avoid the obvious things a critic might identify as an easy dismissal of your findings.

17.3 Presentation

Your results have finished running…what now?

- Understand the effects of all of the factors manipulated in the simulation
- Develop evidence that addresses your research questions

Three approaches to analysis and presentation:

1. Tabulation
2. Visualization
3. Modeling

17.4 Tabulation

- Traditionally, simulation study results are presented in big tables. Tables are fine if…
  - they involve only a few numbers, and a few targeted comparisons
  - it is important to report *exact* values for some quantities
- But simulations usually produce lots of numbers, and involve making lots of comparisons.
  - relative performance of alternative estimators
  - performance under different conditions for the data-generating model
- Exact values for bias/RMSE/type-I error are not usually of interest. And in fact, we rarely have them due to Monte Carlo simulation error.
- It is often more useful and insightful to present results in graphs (Gelman, Pasarica, & Dodhia, 2002).

17.5 Visualization

Visualization should nearly always be the first step in analyzing simulation results.

This often requires creating a *BUNCH* of graphs to look at different aspects of the data.

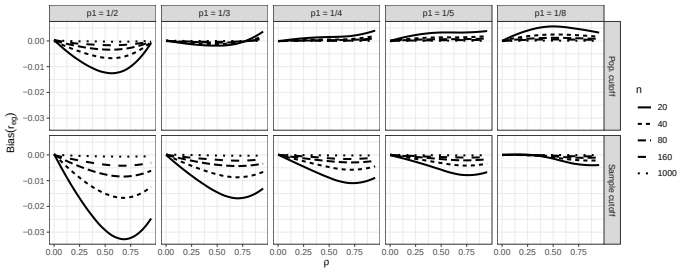
Helpful tools/concepts:

- Boxplots are often useful for depicting range and central tendency across many combinations of parameter values.
- Use color, shape, and line type to encode different factors
- Small multiples (faceting) can then encode further factors (e.g., varying sample size)

17.5.1 Example 1: Biserial correlation estimation

Our first example shows the bias of a biserial correlation estimate from an extreme groups design. This simulation was a $96 \times 2 \times 5 \times 5$ factorial design (true correlation for a range of values, cut-off type, cut-off percentile, and sample size). The correlation, with 96 levels, forms the x-axis, giving us nice performance curves. We use line type for the sample size, allowing us to easily see how bias collapses as sample size increases. Finally, the facet grid gives our final factors of cut-off type and cut-off percentile. All our factors, and near 5000 explored simulation scenarios, are visible in a single plot.

```
## `geom_smooth()` using formula 'y ~ x'
```

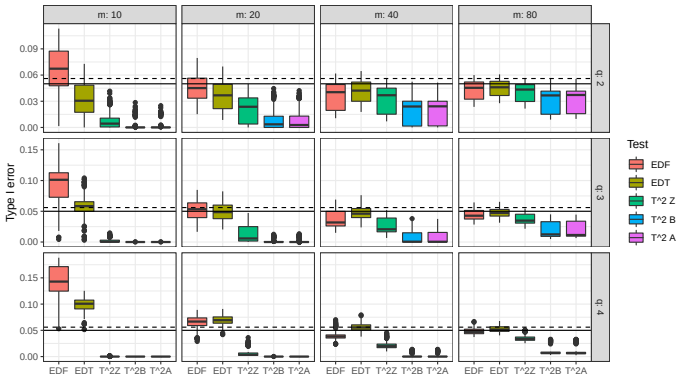


Source: Pustejovsky, J. E. (2014). Converting from d to r to z when the design uses extreme groups, dichotomization, or experimental control. Psychological Methods, 19(1), 92-112.

Note that in our figure, we have smoothed the lines with respect to `rho` using `geom_smooth()`. This is a nice tool for taking some of the simulation jitter out of an analysis to show overall trends more directly.

17.5.2 Example 2: Variance estimation and Meta-regression

- Type-I error rates of small-sample corrected F-tests based on cluster-robust variance estimation in meta-regression
- Comparison of 5 different small-sample corrections
- Complex experimental design, varying
  - sample size (m)
  - dimension of hypothesis (q)
  - covariates tested
  - degree of model mis-specification

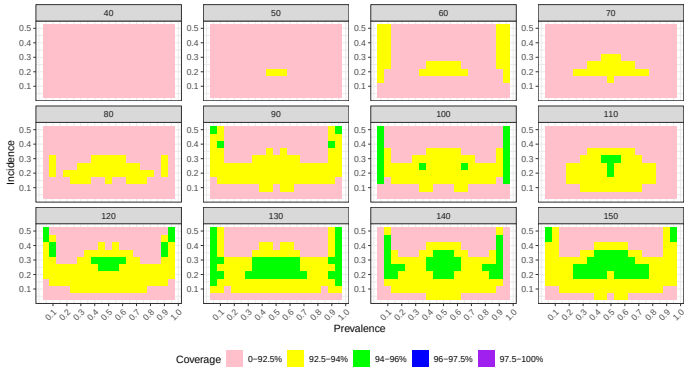


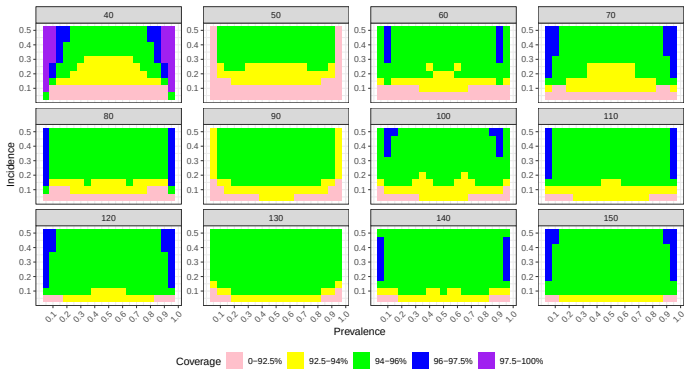
Source: Tipton, E., & Pustejovsky, J. E. (2015). Small-sample adjustments for tests of moderators and model fit using robust variance estimation in meta-regression. *Journal of Educational and Behavioral Statistics, 40*(6), 604-634.

17.5.3 Example: Heat maps of coverage

The visualization below shows the coverage of parametric bootstrap confidence intervals for momentary time sampling data In this simulation study the authors were comparing maximum likelihood estimators to posterior mode (penalized likelihood) estimators of prevalence. We have a 2-dimensional parameter space of prevalence (19 levels) by incidence (10 levels). We also have 15 levels of sample size.

One option here is to use a heat map, showing the combinations of prevelance and incidence as a grid for each sample size level. We break coverage into ranges of interest, with green being "good" (near 95%) and yellow being "close" (92.5% or above). For this to work, we need our MCSE to be small enough that our coverage is estimated precisely enough to show structure.





To see this plot IRL, see Pustejovsky, J. E., & Swan, D. M. (2015). Four methods for analyzing partial interval recording data, with application to single-case research. *Multivariate Behavioral Research, 50*(3), 365-380.

17.6 Modeling

Simulations are designed experiments, often with a full factorial structure. We can therefore leverage classic means for analyzing such full factorial experiment. In particular, we in effect model how a performance measure varies as a function of the different experimental factors. We can use regression or other modeling to do this.

First, in the language of a full factor experiment, we might be interested in the "main effects" or "interaction effects." A main effect is whether, averaging across the other factors in our experiment, a factor of interest systematically impacts our peformance measure. When we look at a main effect, the other factors help ensure our main effect is generalizable: if we see a trend when we average over the other varying aspects, then we can state that for a host of simulation contexts, grouped by levels of our main effect, we see a trend.

For example, consider the Bias of biserial correlation estimate from an extreme groups design example from above. Visually, we see that most factors appear to matter for bias, but we might want to get a sense of how much. In particular, does the the population vs sample cutoff option matter, on average, for bias?

```
options(scipen = 5)
mod = lm( bias ~ fixed + rho + I(rho^2) + p1 + n, data = r_F)
summary(mod, digits=2)
```

```
##
## Call:
## lm(formula = bias ~ fixed + rho + I(rho^2) + p1 + n, data = r_F)
##
## Residuals:
##        Min         1Q     Median         3Q        Max
## -0.0215935 -0.0013608  0.0003823  0.0015677  0.0081802
##
## Coefficients:
##                        Estimate  Std. Error t value Pr(>|t|)
## (Intercept)          0.00218473  0.00015107  14.462  < 2e-16 ***
## fixedSample cutoff -0.00363520  0.00009733 -37.347  < 2e-16 ***
## rho                -0.00942338  0.00069578 -13.544  < 2e-16 ***
## I(rho^2)            0.00720857  0.00070868  10.172  < 2e-16 ***
## p1.L                0.00461700  0.00010882  42.426  < 2e-16 ***
## p1.Q               -0.00160546  0.00010882 -14.753  < 2e-16 ***
```

```
## p1.C                0.00081464  0.00010882   7.486 8.41e-14 ***
## p1^4               -0.00011190  0.00010882  -1.028   0.3039
## n.L                 0.00362949  0.00010882  33.352  < 2e-16 ***
## n.Q                -0.00103981  0.00010882  -9.555  < 2e-16 ***
## n.C                 0.00027941  0.00010882   2.568   0.0103 *
## n^4                 0.00001976  0.00010882   0.182   0.8559
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.003372 on 4788 degrees of freedom
## Multiple R-squared:  0.5107, Adjusted R-squared:  0.5096
## F-statistic: 454.4 on 11 and 4788 DF,  p-value: < 2.2e-16
```

The above printout gives main effects for each factor, averaged across other factors. It is automatically generating linear, quadradic, cubic and fourth order contrasts for the ordered factors of p1 and n. We see that, across other contexts, the sample cutoff is around 0.004 lower than population.

We next discuss two additional tools:

- ANOVA can be useful for understanding major sources of variation in simulation results (e.g., identifying which factors have negligible/minor influence on the bias of an estimator).
- Smoothing (e.g., local linear regression) over continuous factors

```
anova_table <- aov(bias ~ rho * p1 * fixed * n, data = r_F)
summary(anova_table)
```

```
##                   Df   Sum Sq  Mean Sq  F value Pr(>F)
## rho                1 0.002444 0.002444  1673.25 <2e-16 ***
## p1                 4 0.023588 0.005897  4036.41 <2e-16 ***
## fixed              1 0.015858 0.015858 10854.52 <2e-16 ***
## n                  4 0.013760 0.003440  2354.60 <2e-16 ***
## rho:p1             4 0.001722 0.000431   294.71 <2e-16 ***
## rho:fixed          1 0.003440 0.003440  2354.69 <2e-16 ***
## p1:fixed           4 0.001683 0.000421   287.98 <2e-16 ***
## rho:n              4 0.002000 0.000500   342.31 <2e-16 ***
## p1:n              16 0.019810 0.001238   847.51 <2e-16 ***
## fixed:n            4 0.013359 0.003340  2285.97 <2e-16 ***
## rho:p1:fixed       4 0.000473 0.000118    80.87 <2e-16 ***
## rho:p1:n          16 0.001470 0.000092    62.91 <2e-16 ***
## rho:fixed:n        4 0.002929 0.000732   501.23 <2e-16 ***
## p1:fixed:n        16 0.001429 0.000089    61.12 <2e-16 ***
## rho:p1:fixed:n    16 0.000429 0.000027    18.36 <2e-16 ***
## Residuals       4700 0.006866 0.000001
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
library(lsr)
etaSquared(anova_table)
```

```
##                     eta.sq eta.sq.part
## rho            0.021971037  0.26254289
## p1             0.212004203  0.77453319
## fixed          0.142527898  0.69783705
## n              0.123670355  0.66710072
## rho:p1         0.015479114  0.20052330
## rho:fixed      0.030918819  0.33377652
## p1:fixed       0.015125570  0.19684488
## rho:n          0.017979185  0.22560369
## p1:n           0.178055588  0.74260975
## fixed:n        0.120065971  0.66049991
## rho:p1:fixed   0.004247472  0.06439275
## rho:p1:n       0.013216569  0.17638308
## rho:fixed:n    0.026326074  0.29902214
## p1:fixed:n     0.012839790  0.17222072
## rho:p1:fixed:n 0.003857877  0.05883389
```

Perhaps we need an example where some things don't matter? We need to discuss what one learns from this table.—Miratrix

17.7 Presentation

- Present selected results that clearly illustrate the main findings from the study and anything unusual/anomolous.
- In the text of your write-up, include examples that make specific numerical comparisons.
- Include supplementary materials containing
  - additional figures and analysis
  - complete simulation results
  - reproducible code for running the simulation and doing the analysis

Chapter 18

Simulation under the Potential Outcomes Framework

If we are in the business of evaluating how various methods such as matching or propensity score weighting work in practice, we would probably turn to the potential outcomes framework for our simulations. The potential outcomes framework is a framework typically used in the causal inference literature to make very explicit statements regarding the mechanics of causality and the associated estimands one might target when estimating causal effects. While we recommend reading, for a more thourough overview, either [CITE Raudenbush or Field Experiments textbook], we briefly outline this framework here to set out our notation.

Take a sample of experimental units, indexed by i. For each unit, we can treat it or not. Denote treatment as $Z_i = 1$ for treated or $Z_i = 0$ for not treated. Now we imagine each unit has two potential outcomes being the outcome we would see if we treated it ($Y_i(1)$) or if we did not ($Y_i(0)$). Finally, our observed outcome is then

$$Y_i^{obs} = Z_i Y_i(1) + (1 - Z_i) Y_i(0).$$

For a unit, the treatment effect is $\tau_i = Y_i(1) - Y_i(0)$; it is how much our outcome changes if we treat vs. not treat. Frustratingly, for each unit we can only see one of its two potential outcomes, so we can never get an estimate of these individual τ_i. Under this view, causality is a missing data problem: if we only were able to impute the missing potential outcomes, we could have a dataset where we could calculate any estimands we wanted. E.g., the true average treatment effect *for the sample* $\mathcal{S}$ would be:

$$ATE_{\mathcal{S}} = \frac{1}{N}\sum_i Y_i(1) - Y_i(0).$$

The average proportion increase, by contrast, would be

$$API_{\mathcal{S}} = \frac{1}{N}\sum_i \frac{Y_i(1)}{Y_i(0)}$$

18.1 Finite vs. Superpopulation inference

Consider a sample of n units, $\mathcal{S}$, along with their set of potential outcomes. We can talk about the true ATE of the sample, or, if we thought of the sample as being drawn from some larger population, we could talk about the true ATE of that larger population.

This is a tension that often arises in potential outcomes based simulations: if we are focused on $ATE_{\mathcal{S}}$ then for each sample we generate, our estimand could be (maybe only slightly) different, depending on whether our sample has more or fewer units with high τ_i. If, on the other hand, we are focused on where the units came from (which is our data generating model), our estimand is a property of the DGP, and would be the same for each sample generated.

The catch is when we calculate our performance metrics, we now have two possible targets to pick from. Furthermore, if we are targeting the superpopulation ATE, then our error in estimation may be due in part to the representativeness of the sample, *not* the estimation or uncertainty due to the random assignment.

We will follow this theme throughout this chapter.

18.2 Data generation processes for potential outcomes

If we want to write a simulation using the potential outcomes framework, it is clear and transparent to first generate a complete set of potential outcomes, then generate a random assignment based on some assignment mechanism, and finally generate the observed outcomes as a function of assignment and original potential outcomes.

For example, we might say that our data generation process is as follows: First

generate each unit $i = 1, \ldots, n$, as

$$\begin{aligned} X_i &\sim exp(1) - 1 \\ Y_i(0) &= \beta_0 + \beta_1 X_i + \epsilon_i \text{ with } \epsilon_i \sim N(0, \sigma^2) \\ \tau_i &= \tau_0 + \tau_1 X_i + \alpha u_i \text{ with } u_i \sim t_{df} \\ Y_i(1) &= Y_i(0) + \tau_i \end{aligned}$$

with $exp(1)$ being the standard exponential distribution and t_{df} being a t distribution with df degrees of freedom. We subtract 1 from X_i to zero-center it (it is often convenient to have zero-centered covariates so we can then, e.g., interpret τ_0 as the true superpopulation ATE of our experiment).

The above model is saying that we first, for each unit, generate a covariate. We then generate our two potential outcomes. I.e., we are generating what the outcome would be for each unit if it were treated and if it were not treated. We are driving both the level and the treatment effect with X_i, assuming β_1 and τ_1 are non-zero.

One advantage of generating all the potential outcomes is we can then calculate the finite-sample estimands such as the true average treatment effect for the generated sample: we just take the average of $Y_i(1) - Y_i(0)$ for our sample.

Here is some code to illustrate the first part of the data generating process (we leave treatment assignment to later):

```
gen_data <- function( n = 100,
                      R2 = 0.5,
                      beta_0 = 0, beta_1 = 1,
                      tau_0 = 1, tau_1 = 1,
                      alpha = 1, df = 3 ) {
  stopifnot( R2 >= 0 && R2 < 1 )
  X_i = rexp( n, rate = 1 ) - 1
  beta_1 = sqrt( 1 - R2 )
  sigma_e = sqrt( R2 )
  Y0_i = beta_0 + beta_1 * X_i + rnorm( n, sd=sigma_e )
  tau_i = tau_0 + tau_1 * X_i + alpha * rt( n, df = df )
  Y1_i = Y0_i + tau_i

  tibble( X = X_i, Y0 = Y0_i, Y1 = Y1_i )
}
```

And now we see our estimand can change:

```
set.seed( 40454 )
d1 <- gen_data( 50 )
mean( d1$Y1 - d1$Y0 )
```

```
## [1] 0.6374925
```

```
d2 <- gen_data( 50 )
mean( d2$Y1 - d2$Y0 )
```

```
## [1] 0.5479788
```

In reviewing our code, we know our superpopulation ATE should be `tau`, or 1 exactly. If our estimate for `d1` is 0.6 do we say that is close or far from the target? From a finite sample performance approach, we nailed it. From superpopulation, less so.

Also in looking at the above, there are a few details to call out:

- We can store the latent, intermediate quantities (both potential outcomes, in particular) so we can calculate the estimands of interest or learn about our data generating process. When we hand the data to an estimator, we would not provide this "secret" information.
- We are using a trick to index our DGP by an R2 value rather than coefficients on X so we can have a standardized control-side outcome (the expected variation of $Y_i(0)$ will be 1). The treatment outcomes will have more variation due to the heterogeniety of the treatment impacts.
- If we were generating data with a constant treatment impact, then $ATE_S = ATE$ always; this is typical for many similations in the literature. That being said, treatment variation is what causes a lot of methods to fail and so having simulations that have this variation is usually important.

Once we have our *schedule of potential outcomes*, we would then generate the *observed outcomes* by assigning our (synthetic, randomly generated) n units to treatment or control. For example, say we wanted to simulate an observational context where treatment was a function of our covariate. We could model each unit as flipping a weighted coin with some probability that was a function of X_i as so:

$$
\begin{aligned}
p_i &= logit^{-1}(\xi_0 + \xi_1 X_i) \\
Z_i &= Bern(p_i) \\
Y_i &= Z_i Y_i(1) + (1 - Z_i) Y_i(0)
\end{aligned}
$$

Here is code for assigning our data to treatment and control:

```
assign_data <- function( dat,
                         xi_0 = -1, xi_1 = 1 ) {
  n = nrow(dat)
```

```
  dat = mutate( dat,
                p = arm::invlogit( xi_0 + xi_1 * X ),
                Z = rbinom( n, 1, prob=p ),
                Yobs = ifelse( Z == 1, Y1, Y0 ) )
  dat
}
```

We can then add our assignment variable to our given data as so:

```
assign_data( d2 )
```

```
## # A tibble: 50 x 6
##           X     Y0      Y1     p     Z    Yobs
##       <dbl>  <dbl>   <dbl> <dbl> <int>   <dbl>
##  1  0.670    0.667   2.58  0.418     1   2.58
##  2  0.371    0.314   4.57  0.348     1   4.57
##  3  1.94     1.29    3.03  0.719     0   1.29
##  4 -0.244    0.119 -10.0   0.224     1 -10.0
##  5  0.00850  1.44    2.88  0.271     0   1.44
##  6  1.41     1.14    5.02  0.600     1   5.02
##  7 -0.864    0.461   0.802 0.134     1   0.802
##  8 -0.00533 -0.914  -1.17  0.268     0  -0.914
##  9 -0.907   -0.202   0.555 0.129     1   0.555
## 10 -0.363   -0.141   1.16  0.204     1   1.16
## # ... with 40 more rows
```

Note how `Yobs` is, depending on `Z`, either `Y0` or `Y1`. Separating our our DGP and our random assignment underscores the potential outcomes framework adage of the data are what they are, and we the experimenters (or nature) is randomly assigning these whole units to various conditions and observing the consequences.

In general, we might instead put the `p_i` part of the model in our code generating the outcomes, if we wanted to view the chance of treatment assignment as inherent to the unit (which is what we usually expect in an observational context).

18.3 Finite sample performance measures

Let's generate a single dataset with our DGP from above, and run a small experiment where we actually randomize units to treatment and control:

```
n = 100
set.seed(442423)
dat = gen_data(n, tau_1 = -1)
dat = mutate( dat,
              Z = 0 + (sample( n ) <= n/2),
              Yobs = ifelse( Z == 1, Y1, Y0 ) )
mod = lm( Yobs ~ Z, data=dat )
coef(mod)[["Z"]]
```

```
## [1] 0.8914992
```

We can compare this to the true finite-sample ATE:

```
mean( dat$Y1 - dat$Y0 )
```

```
## [1] 1.154018
```

Our finite-population simulation would be:

```
rps <- rerun( 1000, {
  dat = mutate( dat,
              Z = 0 + (sample( n ) <= n/2),
              Yobs = ifelse( Z == 1, Y1, Y0 ) )
  mod = lm( Yobs ~ Z, data=dat )
  tibble( ATE_hat = coef(mod)[["Z"]],
          SE_hat = arm::se.coef(mod)[["Z"]] )
  }) %>%
  bind_rows()

rps %>% summarise( EATE_hat = mean( ATE_hat ),
                   SE = sd( ATE_hat ),
                   ESE_hat = mean( SE_hat ) )
```

```
## # A tibble: 1 x 3
##   EATE_hat    SE ESE_hat
##      <dbl> <dbl>   <dbl>
## 1     1.16 0.248   0.307
```

We are simulating on a single dataset. In particular, our set of potential outcomes is entirely fixed; the only source of randomness (and thus the randomness behind our SE) is the random assignment. Now this opens up some room for critique: what if our single dataset is non-standard?

Our super-population simulation would be, by contrast:

```
rps_sup <- rerun( 1000, {
  dat = gen_data(n)
  dat = mutate( dat,
                Z = 0 + (sample( n ) <= n/2),
                Yobs = ifelse( Z == 1, Y1, Y0 ) )
  mod = lm( Yobs ~ Z, data=dat )
  tibble( ATE_hat = coef(mod)[["Z"]],
          SE_hat = arm::se.coef(mod)[["Z"]] )
  }) %>%
  bind_rows()

rps_sup %>% summarise( EATE_hat = mean( ATE_hat ),
                       SE = sd( ATE_hat ),
                       ESE_hat = mean( SE_hat ))
```

```
## # A tibble: 1 x 3
##   EATE_hat    SE ESE_hat
##      <dbl> <dbl>   <dbl>
## 1     1.00 0.381   0.378
```

First, note our superpopulation simulation is not biased for the superpopulation ATE. Also note the true SE is larger than our finite-sample simulation; this is because part of the uncertainty in our estimator is the uncertainty of whether our sample is representative of the superpopulation.

Finally, this clarifies that our linear regression estimator is estimating standard errors assuming a superpopulation model. The true finite sample standard error is less than the expected estimated error: from a finite sample perspective, our estimator is giving overly conservative uncertainty estimates. (This discrepancy is often called the correlation of potential outcomes problem.)

18.4 Nested finite simulation procedure

We just saw a difference between a specific, single, finite-sample dataset and a superpopulation. What if we wanted to know if this phenomenon was more general across a set of datasets? This question can be levied more broadly: if we run a simulation on a single dataset, this is even more narrow than running on a single scenario: if we compare methods and find one is superior to another for our single dataset, how do we know this is not an artifact of some specific characteristic of *that data* and not a general phenomonen at all?

One way forward is to run a nested simulation, where we generate a series of finite sample datasets, and then for each dataset run a small simulation. We then calculate the expected finite sample performance across the datasets. One

could almost think of the datasets themselves as a "factor" in our multifactor experiment. This is what we did in [CITE estimands paper]

Borrowing from the simulation appendix of [CITE estimands paper], repeat R times:

1. Generate a dataset using a particular DGP. This data generation is the "sampling step" for a superpopulation (SP) framework. The DGP represents an infinite superpopulation. Each dataset includes, for each observation, the potential outcome under treatment or control.
2. Record the true finite-sample ATE, both person and site weighted.
3. Then, three times, do a finite simulation as follows:

a. Randomize units to treatment and control.
b. Calculate the corresponding observed outcomes.
c. Analyze the results using the methods of interest, recording both the point estimate and estimated standard error for each.

Having only three trials will give a poor estimate of within-dataset variability for each dataset, but the average across the R datasets in a given scenario gives a reasonable estimate of expected variability across datasets of the type we would see given the scenario parameters.

To demonstrate we first make a mini-finite sample driver:

```
one_finite_run <- function( R0 = 3, n = 100, ... ) {
  dat = gen_data( n = n, ... )
  rps <- rerun( R0, {
         dat = mutate( dat,
                       Z = 0 + (sample( n ) <= n/2),
                       Yobs = ifelse( Z == 1, Y1, Y0 ) )
        mod = lm( Yobs ~ Z, data=dat )
        tibble( ATE_hat = coef(mod)[["Z"]],
                SE_hat = arm::se.coef(mod)[["Z"]] )
    }) %>%
    bind_rows()
  rps$ATE = mean( dat$Y1 - dat$Y0 )
  rps
}
```

This driver also stores the finite sample ATE for future reference:

```
one_finite_run()
```

```
## # A tibble: 3 x 3
##   ATE_hat SE_hat   ATE
##     <dbl>  <dbl> <dbl>
## 1   0.348  0.421 0.768
## 2   1.32   0.472 0.768
## 3   1.17   0.549 0.768
```

We then run a bunch of finite runs.

```
runs <- rerun( 500, one_finite_run() ) %>%
  bind_rows( .id = "runID" )
```

We use `.id` because we will need to separate out each finite run and analyze separately, and then aggregate.

Each finite run is a very noisy simulation for a fixed dataset. This means when we calculate performance measures we have to be careful to avoid bias in the calculations; in particular, we need to focus on estimating SE^2 across the finite runs, not SE, to avoid the bias caused by having a few observations with every estimate.

```
fruns <- runs %>% group_by( runID ) %>%
  summarise( EATE_hat = mean( ATE_hat ),
             SE2 = var( ATE_hat ),
             ESE_hat = mean( SE_hat ),
             .groups = "drop" )
```

And then we aggregate our finite sample runs:

```
res <- fruns %>%
  summarise( EEATE_hat = mean( EATE_hat ),
             EESE_hat = sqrt( mean( ESE_hat^2 ) ),
             ESE = sqrt( mean( SE2 ) ) ) %>%
  mutate( calib = 100 * EESE_hat / ESE )

res
```

```
## # A tibble: 1 x 4
##   EEATE_hat EESE_hat   ESE calib
##       <dbl>    <dbl> <dbl> <dbl>
## 1     0.996    0.380 0.331  115.
```

We see our expected standard error estimate is, across the collection of finite sample scenarios all sharing a similar parent superpopulation DGP, 15% too large for the true expected finite-sample standard error.

We need to keep the squaring. If we look at the SEs themselves, we have further apparent bias due to our *estimated* `ESE_hat` being so unstable due to too few observations:

```
mean( sqrt( fruns$SE2 ) )
```

```
## [1] 0.2944556
```

We can use our collection of mini-finite-sample runs to estimate superpopulation quantities as well. Given that the simulation datasets are i.i.d. draws, we can simply take expectations across all our simulations. The only concern is our estimates of MCSE will be off due to the clustering in our simulation runs.

Here we calculate superpopulation performance measures (both with the squared SE and without; we prefer the squared version):

```
runs %>%
  summarise( EATE_hat = mean( ATE_hat ),
             SE_true = sd( ATE_hat ),
             SE_hat = mean( SE_hat ),
             SE2_true = var( ATE_hat ),
             SE2_hat = mean( SE_hat^2 ) ) %>%
  pivot_longer( cols = c(SE_true:SE2_hat ),
                names_to = c( "estimand", ".value" ),
                names_sep ="_" ) %>%
  mutate( inflate = 100 * hat / true )
```

```
## # A tibble: 2 x 5
##   EATE_hat estimand  true   hat inflate
##      <dbl> <chr>    <dbl> <dbl>   <dbl>
## 1    0.996 SE       0.389 0.377    96.9
## 2    0.996 SE2      0.151 0.142    93.9
```

18.5 Calibrated simulations and the potential outcomes framework

The potential outcomes framework provides a natural way of generating calibrated simulations [CITE Stuart]. Calbirated simulations are simulations tailored to specific real-world scenarios, to maximize their face validity as being representative of something we would see in practice. One way to generate a calibrated simulation is to use an existing datasets to generate plausible scenarios.

One way to do this with potential outcomes is to take an existing randomized experiment or observational study and impute all the missing potential outcomes under some specific scheme. This fully defines the sample of interest and thus any target parameters, such as a measure of heterogeneity, are then known. We will then synthetically, and repeatedly, randomize and "observe'' outcomes to be analyzed with the methods we are testing. We could also resample from our dataset to generate datasets of different size, or to have a superpopulation target as our estimand.

The key feature is the imputation step. One baseline method one can use is to generate a matched-pairs dataset by, for each unit, finding a close match given all the demographic and other covariate information of the sample. We then use the matched unit as the imputed potential outcome.
By doing this (with replacement) for all units we can generate a fully imputed dataset which we then use as our population. This can preserve complex relationships in the data that are not model dependent. In particular, if outcomes tend to be coarsely defined (e.g., on an integer scale) or have specific clumps (such as zero-inflation or rounding), this structure will be preserved.

One concern with this approach is the noise in the matching will in general dilute the structure of the treatment effect.
This is akin to measurement error diluting found relationships in linear models. We can then sharpen these relationships towards a given model by first imputing missing outcomes using a specified model, and then matching on all units including the imputed potential outcome. This is not a data analysis strategy, but instead a method of generating synthetic data that both has a given structure of interest and also remains faithful to the idiosyncrasies of an actual dataset.

A second approach that allows for varying the level of a systematic effect is to specify the treatment effect model and impute treatment outcomes for all control units.
Then the complex structure between covariates and $Y(0)$ would be perfectly preserved. Unfortunately, this would give 0 idiosyncratic treatment variation (unit-to-unit variation in the τ_i that is not explained by a model). To add in idiosyncratic variation we would then need to generate a distribution of perturbations and add these to the imputed outcomes just as an error term in a regression model.

Regardless, once a fully observed sample has been obtained we can investigate several aspects of our estimators as above.

Chapter 19

Simulations as evidence

We began this book with an acknowledgement that simulation is fraught with the potential for misuse: *simulations are doomed to succeed*. We close by reiterating this point, and also disussing several ways researchers might argue for their simulations being more useful than typical.

In particular a researcher might do any of the following

1. Use extensive multi-factor simulations
2. Beat them at their own game. Generate simulations based on prior literature.
3. Build calibrated simulations

19.1 Use extensive multi-factor simulations

"If a single simulation is not convincing, use more of them," is one principle a reseacher might take. By conducting extensive multifactor simulations, once can explore a large space of possible data generating scenarios. If, across the full range of scenarios, a general story bears out, then perhaps that will be more convincing than a narrower range.

Of course, the critic will claim that some aspect that is not varying is the real culprit. If this is unrealistic, then the findings, across the board, may be less relevant. Thus, pick the factors one varies with care.

19.2 Beat them at their own game

If a prior paper uses a simulation to make a case, one approach is to replicate that simulation, adding in the new estimator one wants to evaluate. This makes

it (more) clear that you are not fishing: you are using something established in the literature as a published benchmark. By constraining oneself to published simulations, one has less wiggle room to cherry pick a data generating process that works the way you want.

19.3 Calibrated simulations

A practice in increasing vogue is to generate *calibrated simulations*. These are simulations tailored to a specific applied contexts, with the results of the simulation studies designed to more narrowly inform what assumptions and structures are necessary in order to make progress in that context.

Often these simulations are built out of existing data. For example, one might sample, with replacement, from the covariate distribution of an actual dataset so that the distribution of covariates is authentic in how the covariates are distributed and, more importantly, how they co-relate.

The problem with this approach is one still needs to generate a ground truth to assess how well the estimators work in practice. It is very easy to accidently put a very simple model in place for this component, thus making a calibrated simulation quite naive in the very way that counts.

Chapter 20

The Parametric bootstrap

An inference procedure very much connected to simulation studies is the parametric bootstrap. The parametric bootstrap is a bootstrap technique designed to obtain standard error estimates for an estimated parametric model. It can do better than the case-wise bootstrap in some circumstances, usually when there is need to avoid the discrete, chunky nature of a casewise bootstrap (which will only give values that exist in the original dataset).

For a parametric bootstrap, the core idea is to fit a given model to actual data, and then take the parameters we estimate from that model as the DGP parameters in a simulation study. The parametric bootstrap is a simulation study for a specific scenario, and our goal is to assess how variable (and, possibly, biased) our estimator is for this specific scenario. If the behavior of our estimator in our simulated scenario is similar to what it would be under repeated trials in the real world, then our bootstrap answers will be informative as to how well our original estimator performs in practice. This is the bootstrap principle, or analogy with an additional assumption that the real-world is effectively well specified as the parameteric model we are fitting.

In particular we do the following:

1. generate data from a model with coefficients as estimated on the original data.
2. repeatedly estimate our target quantity on a series of synthetic data sets, all generated from this model.
3. examine this collection of estimates to assess the character of the estimates themselves, i.e. how much they vary, whether we are systematically estimating too high or too low, and so forth.
4. The variance and bias of our estimates in our simulation is probably like the actual variance and bias of our original estimate (this is precisely the bootstrap analogy).

A key feature of the parametric bootstrap is it is not, generally, a multifactor simulation experiment. We fit our model to the data, and use our best estimate of the world, as given by the fit model, to generate our data. This means we generally want to simulate in contexts that are (mostly) *pivotal*, meaning the distribution of our test statistic or point estimate is relatively stable across different scenarios. In other words, we want the uncertainty of our estimator to not heavily depend on the exact parameter values we use in our simulation, so that if we are simulating with incorrect parameters our bootstrap analogy will still hold.

Often, to achieve a reasonable claim of being pivotal, we will focus on standardized statistics, such as the t-statistic of

$$t = \frac{est}{\widehat{SE}}$$

It is more common for the distribution of a standardized test statistic to have a canonical distribution across scenarios than an absolute estimate.

20.1 Air conditioners: a stolen case study

Following the case study presented in [CITE bootstrap book], consider some failure times of air conditioning units:

```
dat = c( 3, 5, 7, 18, 43, 85, 91, 98, 100, 130, 230, 487 )
```

We are interested in the log of the average failure time:

```
n = length(dat)
y.bar = mean(dat)
theta.hat = log( y.bar )

c( n = n, y.bar = y.bar, theta.hat = theta.hat )
```

```
##          n      y.bar  theta.hat
##  12.000000 108.083333   4.682903
```

We are interested in this because we are modeling the failure time of the air conditioners with an exponential distribution. This means we will generate new failure times with an exponential distribution:

```
reps = replicate( 10000, {
    smp = rexp(n, 1/y.bar)
    log( mean( smp ) )
})

res_par = tibble(
  bias.hat = mean( reps ) - theta.hat,
  var.hat = var( reps ),
  CIlog_low = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.975),
  CIlog_high = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.025),
  CI_low = exp( CIlog_low ),
  CI_high = exp( CIlog_high ) )
res_par
```

```
## # A tibble: 1 x 6
##   bias.hat var.hat CIlog_low CIlog_high CI_low CI_high
##      <dbl>   <dbl>     <dbl>      <dbl>  <dbl>   <dbl>
## 1  -0.0420  0.0856      4.07       5.21   58.4    184.
```

Note how we are, as usual, in our standard simulation framework of repeatidly (1) generating data and (2) analyzing the simulated data. Nothing is changed.

The nonparametric, or case-wise, bootstrap (this is what people normally mean when they say bootstrap) would look like this:

```
reps = replicate( 10000, {
    smp = sample( dat, replace=TRUE )
    log( mean( smp ) )
})

res_np = tibble(
  bias.hat = mean( reps ) - theta.hat,
  var.hat = var( reps ),
  CIlog_low = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.975),
  CIlog_high = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.025),
  CI_low = exp( CIlog_low ),
  CI_high = exp( CIlog_high ) )


bind_rows( parametric = res_par,
           casewise = res_np, .id = "method") %>%
  mutate( length = CI_high - CI_low )
```

```
## # A tibble: 2 x 8
```

```
##   method     bias.hat var.hat CIlog_low CIlog_high CI_low CI_high length
##   <chr>         <dbl>   <dbl>     <dbl>      <dbl>  <dbl>   <dbl>  <dbl>
## 1 parametric  -0.0420  0.0856      4.07       5.21   58.4    184.   126.
## 2 casewise    -0.0651  0.132       3.90       5.33   49.6    207.   157.
```

This is *also* a simulation: our data generating process is a bit more vague, however, as we are just resampling the data. This means our estimands are not as clearly specified. For example, in our parameteric approach, our target parameter is known to be true. In the case-wise, the connection between our DGP and the parameter `theta.hat` is less explicit.

Overall, in this case, our parametric bootstrap can model the tail behavior of an exponential better than case-wise. Especially considering the small number of observations, it is going to be a more faithful representation of what we are doing–provided our model is well specified for the real world distribution.

Bibliography