Designing Monte Carlo Simulations in R

Luke W. Miratrix and James E. Pustejovsky (Equal authors)

2023-11-20

Contents

Welcome

Monte Carlo simulations are a computational technique for investigating how well something works, or for investigating what might happen in a given circumstance. When we write a simulation, we are able to control how data are generated, which means we can know what the "right answer" is. Then, by repeatedly generating data and then applying some statistical method that data, we can assess how well a statistical method works in practice.

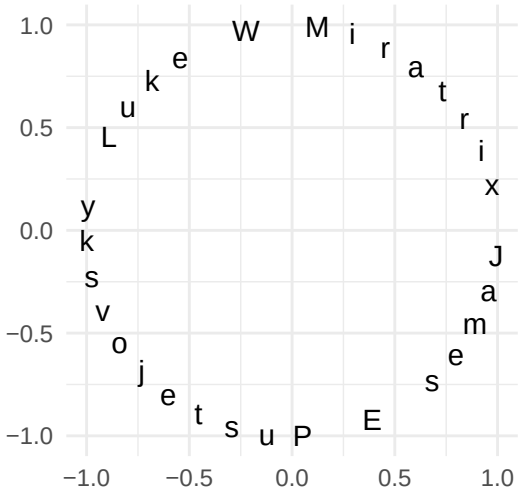
Monte Carlo simulations are an essential tool of inquiry for quantitative methodologists and students of statistics, useful both for small-scale or informal investigations and for formal methodological research. In this monograph, we aim to provide an introduction to the logic and mechanics of designing simulation studies, using the R programming language. Our focus is on simulation studies for formal research purposes (i.e., as might appear in a journal article or dissertation) and for informing the design of empirical studies (e.g., power analysis). That being said, the ideas of simulation are used in many different contexts and for many different problems, and we believe the overall concepts illustrated by these "conventional" simulations readily carry over into all sorts of other types of use, even statistical inference!

Mainly, this book gives practical tools (i.e., lots of code to simply take and repurpose) along with some thoughts and guidance for writing simulations. We hope you find it to be a useful handbook to help you with your own projects, whatever they happen to be!

License

About the authors

We wrote this book in full collaboration, because we thought it would be fun to have some reason to talk about how to write simulations, and we wanted more people to be writing high-quality simulations. Our author order is alphabetical, but perhaps imagine it as a circle, or something with no start or end:



But anymore, more about us.

James E. Pustejovsky is an associate professor at the University of Wisconsin - Madison, where he teaches in the Quantitative Methods Program within the Department of Educational Psychology. He completed a Ph.D. in Statistics at Northwestern University.

Luke Miratrix: I am currently an associate professor at Harvard University's Graduate School of Education. I completed a Ph.D. in Statistics at UC Berkeley after having traveled through three different graduate programs (computer science at MIT, education at UC Berkeley, and then finally statistics at UC Berkeley). I then ended up as an assistant professor in Harvard's statistics department, and moved (back) to Education a few years later.

Over the years, simulation has become a way for me to think. This might be because I am fundamentally lazy, and the idea of sitting down and trying to do a bunch of math to figure something out seems less fun than writing up some code "real quick" so I can see how things operate. Of course, "real quick" rarely is that quick – and before I know it I got sucked into trying to learn some esoteric aspect of how to best program something, and then a few rabbit holes later I may have discovered something interesting! I find simulation quite absorbing, and I also find them reassuring (usually with regards to whether I have correctly implemented some statistical method). This book has been a real pleasure to write, because it's given me actual license to sit down and think

about why I do the various things I do, and also which way I actually prefer to approach a problem. And getting to write this book with my co-author has been a particular pleasure, for talking about the business of writing simulations is rarely done in practice. This has been a real gift, and I have learned so much.

Acknowledgements

The material in this book was initially developed through courses that we offered at the University of Texas at Austin (James) and Harvard University (Luke) as well as from a series of workshops that we offered through the Society for Research on Educational Effectiveness in June of 2021. We are grateful for feedback, questions, and corrections we have received from many students who participated in these courses. Some parts of this book are based on memos or other writings generated for various purposes, some of which were written by others. This is been attributed throughout.

Chapter 1

Introduction

In this text we present an approach for writing Monte Carlo simulations in R. 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. In particular, we try to provide a guide to designing simulation studies to answer questions about statistical methodology.

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

Simulation can be useful across a wide range of areas. 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) when the population variances are not equal. We revisit this example later. 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 practical benefits and costs of using a model that allows for cross-site impact variation for a multi-site trial (**?**)?

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 or robust standard errors 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 (the bootstrap 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. 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 for school assignment where families rank some number of schools 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 exploit this process to estimate the causal impact of being assigned to one school vs. another, treating the lottery as a natural experiment, 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 (**?**).

For another example, one that possibly illustrates the perils of simulation as taking us away from results that pass face validity, **?** simulate the process of firing teachers depending on their estimated value added scores. Using simulation of different levels of draconian policy, they argue that substantial portions of teachers should be fired each year. Here we see a clean example of how the assumptions driving a simulation can be explored, so we can see what the consequences of a system would be… if the assumptions behind the simulation were true.

A 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

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 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. (In fact, this book is in a variant of R Markdown.)

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 cannot. 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), to have multiple tables of data in play at the same time, and to leverage the vast array of other people's work all make R an attractive option.

1.3.1 Templates vs. Patterns

We generally adhere to a simple, modular approach to building simulations. We (repeatidly) demonstrate a set sequence of steps, going from coding a data generating process, to the estimation methods, to the code for evaluating a simulation result, to the final multifactor experiment. But our many case studies are not all precisely the same; the idea of this sequence of steps is paramount, but coding is not a rigid process. Different aspects of a particular problem may call for doing things in a slightly different manner. And often, it is merely an issue of style, or choice.

We believe that the variants of the coding pattens we showcase are just as important as the patterns themselves, as they take us out of a rigid model of thinking about the creation of simulations. The different takes on the same idea will, we hope, expand the sense of what is possible, and also triangulate the core coding principles we are attempting to espouse.

That being said, much of this code can be taken verbatim, tweaked for your own ends, and used. We hope you end up doing just that!

1.3.2 The tidyverse and a recommended text

Layered on top of R are a collection of packages that make data wrangling and management much, much easier. This collection is called the "tidyverse," and much of this book heavily relies on it. Loading the tidyverse packages is straightforward

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

(The second line is to shut up some of tidyverse's weird summarize warnings.) These lines of code are pretty much the header of any script we use.

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 cite portions of this text throughout this manuscript.

1.3.3 Functions

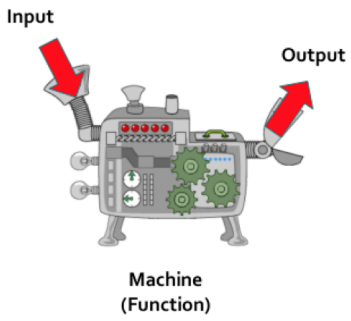


Figure 1.1: A function as a machine

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

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 (see figure above). 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 desired 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. The things we pass to the function, `N` and `mn`, are called *parameters*, or *inputs*. Inside the function, we can use these to make calculations and so forth.

We call our new method as so:

```
one_run( 100, 5 )
```

```
## [1] 2.61271e-74
```

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

```
## [1] 0.4190303
```

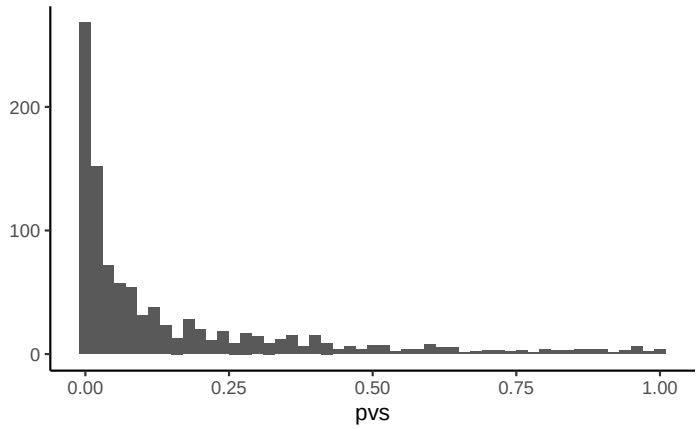
```
one_run( 10, 0.3 )
```

```
## [1] 0.9745024
```

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:

```
sum( pvs <= 0.05 ) / 1000
```

```
## [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.

1.3.4 A dangerous function

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

```
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 operato
```

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
```

1.3.5 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 as documentation for sketching out a general plan of how to organize code.

1.3.6 %>% (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 lowly one-sample t-test under violations of the normality assumption. In particular, we will examine the coverage of our t-test. *Coverage* is the chance of a confidence interval capturing the true parameter value.

The goal of this chapter 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.

Before simulation, we want to understand what we are investigating. 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 high and very rarely does our interval miss because it is entirely too low. 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 rate 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.

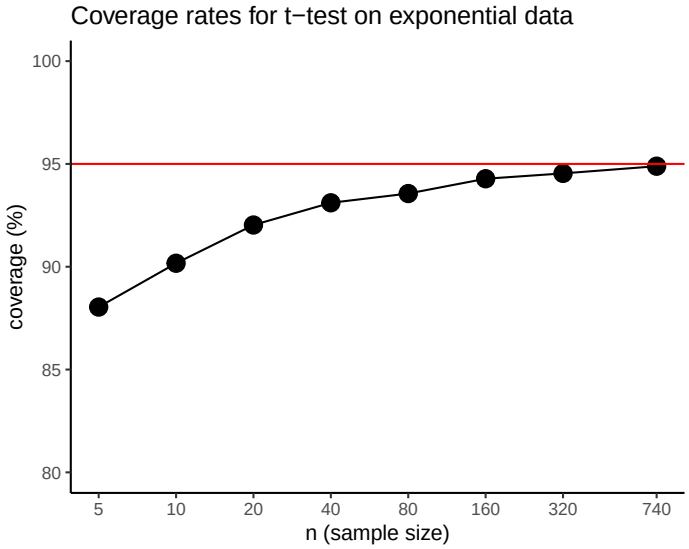
```
run.experiment = function( n ) {
  rps = replicate( 10000, {
    dat = rexp( n )
    tt = t.test( dat )
    findInterval( 1, tt$conf.int )
  })

  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). This is kind of like a for loop, and is the tidyverse form of the `sapply()` method.

```
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, ith a log scale for our x-axis:

```
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 (%)" ) +
  coord_cartesian( ylim=c(80,100) )
```



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

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 some 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.

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, recording R sets of results.
4. **Summarize** the performance of the procedure across our R repetitions to assess how well our estimators work in practice.

We will then repeat the above for different 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.

At a higher level, we 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.

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())
results <- bind_rows(results)

# 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()
```

This function will open up a new R script for you contains a template for a simulation study, with sections corresponding to each component. Starting with this, you will already be well on the road to writing a tidy simulation.

3.2.1 The Data Generating Process (DGP)

The data-generating process 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 (e.g., what the real treatment effect is), so we can later know if our methods are doing the right thing.

3.2.2 Estimation methods

The estimation methods are the sets of statistical procedures under examination. For example, an estimation method might be to estimate the average growth rate along with a standard error. Or it might be conduct a hypothesis test, like we saw earlier.

Each estimation methods should take a dataset and produces a set of estimates or results (i.e., point estimates, standard errors, confidence intervals, p-values, etc.). You probably should have different functions for each estimation method you are investigating.

A well written estimation methods should, in principle, work on real data as well as simulated data; the more we can "black box" our methods into a single function calls, the easier it will be to separate out the structure of the simulation from 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: repeatedly run a specified 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. We usually do this with `bind_rows()` or use methods that do it automatically.

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.

Key point: We also want a dataframe back from this process because we are going to eventually stack the results up to make one long dataframe of results. Happily the `dplyr` package generally gives us dataframes so will not usually be

a problem. But each step of the way, we will be generating data frames to keep things tidy.

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 examine 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 will discuss this after we more fully develop the core concepts listed above (see Chapter **?**).

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, 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)
```

```r
# 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
```

```r
# 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. Do not 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)
```

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 would get a new set of simulated data:

4.1.1 Coding remark

In the above, we built some sample code, and then bundled it into a function by literally cutting and pasting the initial work we did into a function skeleton. In the process, we shifted from having variables in our workspace with different names to using those variable names as parameters in our function call.

Doing this is not without hazards, however. In particular, by the end of this, our workspace has a variable `mu` and our function has a parameter named `mu`. Inside the function, R will use the parameter `mu` first, but this is potentially

confusing. As is, potentially lines like `mu = mu`. What this line means is "the function's parameter called `mu` should be set to the variable called `mu`." These are different things (with the same name).

One way to check your code, once a function is built, to commend out the initial code (or delete it), then restart R or at least clear out the workspace, and then re-run the code that uses the function. If things still work, then you should be somewhat confident you successfully bundled your code into the function.

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 = 3.0065, num df = 3, denom df = 11,
## p-value = 0.07647
```

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] 0.07647251
```

We might instead write that code ourselves. This has some plusses and minuses; see **?**.

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 by hand, 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] 0.1441605
```

4.3 Running the simulation

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_aov` 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 aggregate and analyze.

```
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_aov(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 0.00404 0.0443
```

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

We next call `one_step()` over and over; see **?** for some discussion of options.

```
sim_data <- rerun(4,
                  one_run(mu = mu, sigma_sq = sigma_sq,
                          sample_size = sample_size) )
```

```
## Warning: `rerun()` was deprecated in purrr 1.0.0.
## i Please use `map()` instead.
##   # Previously
## rerun(4, one_run(mu = mu, sigma_sq = sigma_sq,
## sample_size = sample_size))
##
##   # Now
## map(1:4, ~ one_run(mu = mu, sigma_sq = sigma_sq,
## sample_size = sample_size))
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to
## see where this warning was generated.
```

This gives a list of dataframes, one for each `rerun()` call. 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 0.00386 0.0266
## 2 0.00686 0.0874
## 3 0.0118  0.00684
## 4 0.00882 0.0332
```

Voila! Simulated p-values!

4.4 Analyzing the Simulation

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 <- rerun(10000,
  sim_data <- one_run(mu = mu,
                      sigma_sq = sigma_sq,
                      sample_size = sample_size) )
p_vals <- bind_rows(p_vals)
p_vals
```

```
## # A tibble: 10,000 x 2
##     ANOVA Welch
##     <dbl> <dbl>
##  1 0.0582 0.114
##  2 0.784  0.728
##  3 0.0473 0.166
##  4 0.334  0.516
##  5 0.0182 0.284
##  6 0.498  0.650
##  7 0.0155 0.423
##  8 0.261  0.445
##  9 0.815  0.495
## 10 0.343  0.396
## # ... 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.

```
sum(p_vals$ANOVA < 0.05) / 10000
```

```
## [1] 0.1428
```

This is equivalent to taking the mean of the logical conditions:

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

```
## [1] 0.1428
```

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] 0.0659
```

The Welch test does much better, although it appears to be 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 monte carlo simulation error in our studies.

4.5 Exercises

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

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 Table 1?
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). E.g., it might produce results like this:

```
one_run_long()
```

```
## # A tibble: 2 x 2
##   method pvalue
##   <chr>   <dbl>
## 1 ANOVA  0.0158
## 2 Welch  0.116
```

Modify `one_run()` to do this, update your simulation code, 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 quantities 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 specified 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 that define a series of steps. This is especially important for more complex DGPs, such as those for hierarchical data, as we will see later on. These models will often be a series of linear equations that use the same common elements, and possibly build on one another. Our equations will, in the end, be a function of a set of parameters, values that we must specify. We then convert these equations to code by following the steps laid out. In general, we have several components to a full mathematical model that we could use to generate data.

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

In the real world, we don't tend to think of structural covariates as covariates per se, they are more just consequences of the data. We rarely model them, but instead condition on them, in a statistical analysis. In a simulation, however, we will have to decide how they come to be.

MODEL This is the parametric relationship between everything, such as a specification of how the outcomes are linked to the covariates. This includes speci-

fication of the randomness (the distribution of the residuals, etc.). The model will be the list of equations you might see in the analysis of the data you are trying to generate.

We will also have some extra parts of the model, that define how to generate the structural covariates. For example, we might specify that site sizes are uniform between a specified minimum and maximum size.

DESIGN PARAMETERS These are, e.g., the number of sites and the range of allowed site sizes. These will control how we generate the structural covariates.

PARAMETERS These are the specifics: for a given model, parameters describe how strong a relationship there is between covariate and outcome, variance of the residuals, and so forth. We usually estimate these *from* data. Critically, if we know them, we can *generate new data.*

For example, for the Welch data earlier we have, for observation i in group g, a mathematical representation of our data of:

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

These math equations would also come along with specified parameter values (the μ_g, the σ_g^2), and the design parameter of the sample sizes.

5.1 A statistical model is a recipe for data generation

The next step is to 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 last piece is very different from real-world analysis.

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)
}
```

Our function takes parameters as we normally thing of them (`mu`, `sigma_sq`), and other values that we might not think of as parameters per-se (`sample_size`). When simulating data, we have to specify quantities that we, when analyzing data, often have to take for granted.

We our method as so:

```
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 0.547
##  2     1 1.66
##  3     1 1.60
##  4     2 2.16
##  5     2 0.456
##  6     2 2.31
##  7     2 1.85
##  8     2 1.55
##  9     2 2.25
## 10     3 4.53
## 11     3 5.72
## 12     4 5.17
## 13     4 4.92
## 14     4 7.53
## 15     4 6.82
```

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:

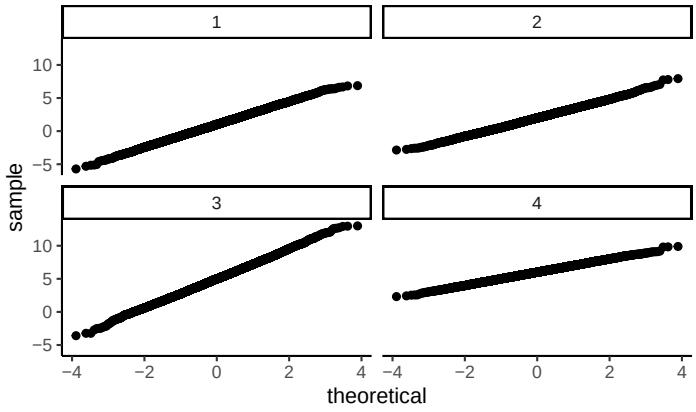
```
check_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                            sample_size = rep(10000, 4))

chk <- check_data %>% group_by( group ) %>%
  dplyr::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  1.01     1  3.02      3
## 2     2 10000  1.98     2  2.02      2
## 3     3 10000  4.98     5  4.97      5
## 4     4 10000  6.00     6  1.00      1
```

We are recovering our parameters.

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 Example: Simulating 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 (where, for example, students are randomized to treatment or control within each of a series of sites) has explored how variation in the size of impacts across sites can cause bad things can happen. In this case study we extend this work to ask what impact treatment variation has for cluster randomized trials.

To set the stage, we identify three different ways one might analyze data from a cluster randomized trial:

- 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 (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. In particular, we want to know if a relationship between site size and site impact would bias any of our three methods.

5.3.1 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 with impact then how we average our sites is going to matter.

Usually, when running a simulation, 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 that 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 should have some impact variation across sites.
- We should be able to connect impact variation to site size to explore a more malicious context.

Overall, our final data should be a collection of clusters with different sizes and different baseline mean outcomes. Some of the clusters will be treated, and some not. We can imagine our final data being individual students, with each student

having a cluster id, a treatment assignment (shared for all in the cluster) and an outcome.

5.3.2 A model for a cluster RCT

It is usually easiest to start a recipe for data generating by writing down the mathematical model. Write down something, and then specify any parts that you are using, in an interative process. For our model, we start with a model for the outcome as a function of any covariates desired.

A covariate in a linear model creates a relationship between that covariate and the outcome. In looking at our goals, this tells us we want to have treatment assignment as a covariate so we can have a treatment impact. To keep things simple, we plan for a common treatment impact within cluster: if we treat a cluster, everyone in the cluster is raised by some specified amount.

We also want the size of impact to possibly vary by site size. This suggests we also want a treatment by site size interaction term. Instead of just using the site size, however, we are going to standardize our site sizes so they are more interpretable. This makes it so if we double the sizes of all the sites, it does not change our size covariate: we want the size covariate to be relative size, not absolute. To do this, we 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 then specify our multilevel 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. A main effect 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.

So far we have a mathematical model analagous to what we would write if we were *analyzing* the data. To *generate* data, we also need several other quantities specified. First, we need to know the number of clusters (J) and the sizes of the clusters (n_j, for $j = 1, ..., J$). We have to provide a recipe for generating these sizes. We might try

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

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. This specification is nice in that we can determine two parametes, $\bar{n}$ and α, to get our site sizes, and both parameters are easy to comprehend: average site size and amount of site size variation.

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

$$\gamma_2 Z_j S_j = \gamma_2 Z_j \left(\frac{n_j - \bar{n}}{\bar{n}} \right) = \gamma_2 Z_j \alpha U_j,$$

where U_j is the $U_j \sim unif(-1, 1)$ uniform variable used to generate n_j. Due to our standardizing by average site size, we make our covariate not change in terms of its importance as a function of site size, but rather as a function of site variation 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 heterogeneiety 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. We will see code for this below.

5.3.3 Converting our model to code

For multilevel data generation, we follow our model, and go by layers. First, we generate the sites:

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

Then we generate the 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.

We start by specifying 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.

Inside the model, we will have a block of code to generate the sites, and then another to generate the students.

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 a treatment and control 0/1 indicator.

There is also a serious error in the above code (serious in that the code will run and look fine in many cases, but not always do what we want); we leave it as

an exercise (see below) to find and fix it.

Make the individuals. We use the site characteristics to then generate the individuals. A key piece here is the `rep()` function that takes a list and repeats each element of the list a specified number of times. Another key

```
  # 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 repeats each number $(1, 2, /ldots, J)$, the corresponding number of times as listed in `nj`.

We put the above code in our function skeleton. When we call our function 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 1  2.35779312
## 2    1 1  0.57601909
## 3    1 1  1.37583246
## 4    2 0 -0.04417977
## 5    2 0  0.35837270
## 6    2 0  2.43903900
## 7    2 0 -1.45205220
## 8    2 0 -1.05747062
## 9    2 0 -0.50122103
## 10   3 0 -0.25961309
## 11   3 0 -0.42459575
```

Our data generation code is complete. The next step is to test the code, making sure it is doing what we think it is (see exercises).

5.3.4 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.)

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

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.

We would call `gen_dat_model` from our usual simulation driver as follows:

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

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

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

  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.

We are transforming our ICC parameter into specific other parameters that are used in our actual model to maintain our effect size interpretation of our simulation. We haven't even modified `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.

5.4 Exercises

5.4.1 The 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]  4.6290066  3.3199535 -0.1054816  2.1896919
## [5]  2.4605947  0.1443433  1.0526101  1.9674069
```

1. Modify the above `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. Simulate a dataset with low degrees of freedom and plot it to see if you see a few outliers.
2. Now generate more data and calculate the standard deviations to see if they are correctly calibrated (generate a big dataset to ensure you get a reliable standard deviation estimate).
3. Once you are satisfied you have a correct DGP function, re-run the Type-I error rate calculations from the prior exercises link on page **?** using a t-distribution with 5 degrees of freedom. Do the results change substantially?

5.4.2 The Cluster RCT DGP

4. 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?
5. 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.

6. Extend the data generating process to include an individual level covariate X that is predictive of outcome. In particular, you will want to adjust your level one equation to

$$Y_{ij} = \beta_{0j} + \beta_1 X_{ij} + \epsilon_{ij}.$$

Keep the same β_1 for all sites. You will have to specify how to generate your X_{ij}. For starters, just generate it as a standard normal, and don't worry about having the mean of X_{ij} vary by sites unless you are excited to try to get that to work.

Chapter 6

Estimation procedures

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

```
estimate <- function(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 **?**, 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 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 = 3.574, num df = 3, denom df = 11,
## p-value = 0.05048
```

```
F_results <- ANOVA_F(sim_data)
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 = 7.7855, num df = 3.0000, denom df =
## 3.0976, p-value = 0.05967
```

```
Welch_results <- Welch_F(sim_data)
all.equal(aov_results$p.value, Welch_results)
```

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

See **?** for a bit more on checking functions, and a side-note about transparency in the writing of papers.

6.2 Checking via simulation

If your estimation procedure truly is new, how would you check it? Well, one obvious answer is simulation!

In principle, for large samples and data generated under the assumptions required by your new procedure, you should have a fairly good sense that it should work and how it should work. It is often the case that as you design your simulation, and then start analyzing the results, you will find your estimators are really not working as planned.

This will usually be due to (at least) three factors: you didn't implement your method correctly, your method is not yet a good idea in the first place, and you don't understand something important about how your method works. You can then debug your code, revise your method, and do some serious thinking to eventually end up with a deeper understanding of method that is a better idea in general, and correctly implemented in all likelihood.

For example, in one research project Luke and other co-authors was working on a way to improve Instrumental Variable (IV) estimation using post-stratification. The idea is to group units based on a covariate that predicts compliance status, and then estimate within each group; hopefully this would improve overall estimation.

In the first simulation, the estimates were full of NAs and odd results because we failed to properly account for what happens when the number of compliers was estimated to be zero. That was table stakes: after repairing that, we still found odd behavior and serious and unexpected bias, which turned out to be due to failing to implement the averaging of the groups step correctly. We fixed the estimator again and re-ran, and found that even when we had a variable that was almost perfectly predictive of compliance, gains were still surprisingly minimal. Eventually we understood that the groups with very few compliers were will so unstable that they ruined the overall estimate. These results inspired us to introduce other estimators that dropped or down-weighted these strata, which gave our paper a deeper purpose and contribution.

Simulation is an iterative process. It is to help you, the researcher, learn about your estimators so you can find a way forward with your work. What you learn then feeds back to the prior research, and you have a cycle that you eventually step off of, if you want to finish your paper. But do not expect it to be a single, well-defined, trajectory.

6.3 Multiple estimation procedures

In **?** (**?**) we introduced a case study of evaluating different procedures for estimating treatment impacts in a cluster randomized trial. As a point of design,

we recommend writing different functions for each estimation method one is planning on evaluating. This makes it easier to plug into play different methods as desired, and also helps generate a code base that is flexible and useful for other purposes. It also, continuing our usual mantra, makes debugging easier: you can focus attention on one thing at a time, and worry less about how errors in one area might propagate to others.

For the cluster RCT context, 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`. This use of different packages for different estimators is quite typical: in many simulations, many of the estimation approaches being considered are usually taken from the literature, and if you're lucky this means you can simply use a package that implements those methods.

We load our libraries at the top of our code:

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

Our three analysis functions are then 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 )
}
```

and 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 )
}
```

Note the `stopifnot` command: putting *assert statements* in your code like this is a good way to guarantee you are not introducing weird and hard-to-track errors in your code. For example, R likes to recycle vectors to make them the right length; if you gave it a wrong length in error, this can be a brutal error to discover. These statements halt your code as soon as something goes wrong, rather than letting that initial wrongness flow on to further work, showing up in odd results that you don't understand later on. See Section **?** for more.

Finally, we write 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 )
```

```
## boundary (singular) fit: see help('isSingular')
```

```
## # A tibble: 3 x 4
##   method ATE_hat SE_hat p_value
##   <chr>    <dbl>  <dbl>   <dbl>
## 1 MLM       1.55  0.758  0.0706
## 2 LR        1.55  0.130  0.0529
## 3 Agg       1.63  0.150  0.0583
```

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.

6.4 Exercises

6.4.1 More Welch and adding the BFF test

Let's continue to explore and tweak 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)
}
```

1. Write a function that implements the Brown-Forsythe F*-test (the BFF* test!) as described on p. 130 of Brown and Forsythe (1974). Call it on a sample dataset to check it.
2. Now 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 of Brown and Forsythe (1974).

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

  # fill in the guts here

  return(pval = pval)
}
```

```
# Further R code here
```

6.4.2 More estimators for the CRT

3. Sometimes you might want to consider two versions of an estimator. For example, in our cluster RCT code above we are using robust standard errors for the linear model estimator. Say we also want to include naive standard error estimates that we get out of the `lm` call.

Extend the `OLS` call to be

```
analysis_OLS <- function( dat, robustSE = TRUE ) {
```

and have the code inside calculate SEs based on the flag. Then modify the `analyze_data()` to include both approaches (you will have to call `analysis_OLS` twice).

Note that, efficiency-wise, this might not be ideal, but clarity-wise it might be considered helpful. Articulate two reasons for this design choice, and articulate two reasons for instead having the `analysis_OLS` method generate both standard errors internally in a single call.

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, if we want to know what percent of the time we would reject the null hypothesis (for a given, specified situation) we could estimate it by seeing how often we do in 1000 trials.

The first step for doing this is to generate that set of estimates. Here we do so for our cluster randomized experiment running example:

```
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()

saveRDS( runs, file = "results/cluster_RCT_simulation.rds" )
```

We save our results to a file for future use; this speeds up our lives since we don't have to constantly re-run our simulation each time we want to explore the results.

Now, because we have only a sample of trials rather than the full distribution, our estimates are merely 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)**. The MCSE is the standard error in our estimate 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. The MCSE is *not* related to the estimators being evaluated; the MCSE is a function of how much simulation we can do. In a simulation study, we could, in theory, know *exactly* how well our estimators do for a given context, if we ran an infinite number of simulations; the MCSE tells us how far we are from this ideal, given how many simulation trials we actually ran. Given a desired MCSE, we could similarly determine how many replications were needed to ensure our performance estimates have a desired level of precision.

7.1 Inference vs. Estimation

There are two general classes of analysis one typically does with data: inference and estimation. To illustrate, we continue to reflect on the question of best practices for analyzing a cluster randomized experiment. For this problem, we are focused on the site-average treatment effect, τ. We can ask whether τ is non-zero (inference), and we can further ask what τ actually is (estimation).

More expanded we have:

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. In particular, for our example, to know if there is evidence that there is a treatment effect at all we would test the null of $H_0 : \tau = 0$.

Estimation is when we estimate the actual average treatment effect of τ. Estimation has two major components, the point estimator and the uncertainty estimator. We generally 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.

Inference and estimation 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 could be different.

7.2 Comparing estimators

We often have different methods for obtaining some estimate, and we often want to know which is best. For example, this comparison is the core question behind our running example of identifying which estimation strategy (aggregation, linear regression, or multilevel modeling) we should generally use when analyzing cluster randomized trial data. The goal of a simulation comparing our estimators would be to identify whether our estimation strategies were different, whether one was superior to the other (and when), and what the salient differences were. To fully understand the trade-offs and benefits, we would examine and compare the properties of our different approaches across a variety of circumstances, and with respect to a variety of metrics of success.

For inference, we first might ask whether our methods are valid, i.e., ask whether the methods work correctly when we test for a treatment effect when there is none. For example, we might wonder whether using multilevel models could open the door to inference problems if we had model misspecification, such as in a scenario 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 our estimators perform in circumstances with a non-zero average treatment effect. Do they reject the null

often, or rarely? How much does using aggregation decrease (or increase?) our chances of rejection? These are questions about power.

For estimation, we would generally be concerned with two things: bias and variance. An estimator is biased if it would generally give estimates that are systematically higher (or lower) than the parameter being estimated in a given scenario. The variance of an estimator would be a measure of 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. Continuing our example from above, we might want to examine how well, for example, the standard errors we get from aggregation work as compared to the standard errors we get out of our linear regression approach.

Finally, we might want to know how well confidence intervals based on our methods work. Do the intervals capture the true estimands with the desired level of accuracy, across the simulation trials? Are the intervals for one method generally shorter or longer than those of another?

7.3 Assessing a point estimator

Assessing the actual properties of a point estimator is generally fairly simple. For a given scenario, we repeatedly generate data and estimate effects. We then take summary measures such as the mean and standard deviation of these repeated trials' estimates to estimate the actual properties of the estimator 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.

To be more formal, 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

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

The square root of this, S_T is the true standard error of our estimator (up to Monte Carlo simulation uncertainty).

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

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

It is important to clarify an important point: the *true standard error* of an estimator $\hat{\tau}$ is the standard deviation of $\hat{\tau}$ across multiple datasets. In practice, we never know this value, but in a simulation we can obtain it as the standard deviation of our simulation trial estimates. People generally, when they say "Standard Error" mean *estimated* Standard Error, ($\widehat{SE}$) which is when one uses the emperical data to estimate SE. For assessing actual properties, we have the true standard error (up to Monte Carlo simulation error).

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 lower 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 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, a nonlinear change of 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.

7.3.1 Example on our Cluster RCT

In our cluster RCT running example, we have three estimation methods that we are applying to our simulated data. We next repeatidly generate our data and obtain our estimates as follows:

It is always wise to start with a single scenario to figure out if your code is working and if your intuition is working.

Once complete, we have the individual results of all our methods applied to each generated dataset. The next step is to evaluate how well the estimators did. Regarding our point estimate, we have these primary questions:

- Is it biased? (bias)
- Is it precise? (standard error)
- Does it predict well? (RMSE)

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

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 ),
    MCSE_bias = sd( ATE_hat - ATE ) / sqrt(R)
  )
```

```
## # A tibble: 3 x 4
##   method mean_ATE_hat    bias MCSE_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.

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. It reflects 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.

Which method has the smallest Root Mean Squared Error?

So far linear regression is not doing well: it has more bias and 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.

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: in our simulation we can know an estimator's 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 using this standard error estimate. To understand if this would work in practice, we would need to evaluate not only the behavior of the estimator itself, but the behavior of these associated things. In other words, we generally not only want to know whether our estimator is doing a good job, but we usually want to know whether we are able to get a good standard error for that estimator as well.

To do this we first compare the expected value of $\widehat{SE}$ (estimated with the average $\widehat{SE}$ across our simulation trials) to the actual SE. This tells us whether our uncertainty estimates are *biased*. We could also examine the standard deviation of $\widehat{SE}$ across trials, which tells us whether our estimates of uncertainty are relatively stable. 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, meaning we compare the estimated standard error to the true standard error as a ratio.

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. We discuss relative performance measures further in Section **?**.

7.4.1 Why not assess the estimate SE 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 because:

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

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.

7.4.2 Assessing SEs for our Cluster RCT simulation

For our standard errors, we have a primary question of: Can we estimate uncertainty well? I.e., are our estimated SEs about right?

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. We interpret this quantity relative to the actual standard error to get how far off, as a percent of the actual standard error, we tend to be.

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

```
## # A tibble: 3 x 6
##   method    SE mean_SEhat  infl sd_SEhat 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
```

The SEs for Agg and MLM appear to be a bit conservative on average. (3 or 4 percentage points too big).

The last column (`stability`) 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 would not be particularly good. Here we see the linear regression is more unstable than the other methods (cluster-robust standard errors are generally unstable, so this is not too surprising). It is a bad day for linear regression.

7.5 Assessing an inferential 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 a true null 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 assess validity we would therefore 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. Power is a much more nebulous concept than validity, because some effects (e.g. large effects) are clearly easier to notice than others. If we are comparing estimators to each other, the overall chance of noticing is less of a concern, because we are typically interested in 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, we 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. 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.

7.5.3 The Rejection Rate

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 the given alternate hypothesis represented by the DGP). 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, using the recorded p-values,

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

For the null, one can also plot the emperical cumulative density function of the p-values; a valid test should give a 45° line as the p-values should be standard uniform in distribution.

7.5.4 Our Cluster RCT Simulation

For our scenario, we generated data with an actual treatment effect. Without further simulation, we therefore could only assess power, not validity. This is easily solved! We simply rerun the above code with `ATE = 0`.

```
tictoc::tic()  # Start the clock!
set.seed( 404044 )
runs_val <-
  purrr::rerun( R, one_run( 0 ) ) %>%
  bind_rows( .id="runID" )
tictoc::toc()

saveRDS( runs_val, file = "results/cluster_RCT_simulation_validity.rds" )
```

Assessing power and validity is exactly the same calculation: we see how often we have a p-value less than 0.05. For power we have:

```
runs %>% group_by( method ) %>%
  summarise( power = mean( p_value <= 0.05 ) )
```

```
## # A tibble: 3 x 2
##   method power
##   <chr>  <dbl>
## 1 Agg    0.376
## 2 LR     0.503
## 3 MLM    0.383
```

For validity:

```
runs_val %>% group_by( method ) %>%
  summarise( power = mean( p_value <= 0.05 ) )
```

```
## # A tibble: 3 x 2
##   method power
##   <chr>  <dbl>
## 1 Agg    0.051
```

```
## 2 LR     0.059
## 3 MLM    0.048
```

The power when there is an effect is not particularly high, and the validity is around 0.05. Note linear regression has notabily higher power… but this is in part due to the invalidity of the test (note the rejection rate is around 6%, rather than the target of 5%). This is also likely due to the bias in estimation. Let's see how everyone performs with confidence intervals, next.

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 its estimated uncertainty.

We typically score a confidence interval along two dimensions, **coverage rate** and **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).

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 r, and let $W_r = B_r - A_r$, all for $r = 1, ..., R$. 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 $\Pr(\theta \leq B)$.

7.6.1 Clusters with Confidence

For our CRT simulation, we first have to calculate confidence intervals, and then assess coverage. We could have used methods such as `confint()` in the estimation approaches; this would be preferred if we wanted more accurately calculated confidence intervals that used t-distributions and so forth to account for the moderate number of clusters.

But if we want to use normal assumption confidence intervals we can calculate them post-hoc:

```
runs %>% mutate( CI_l = ATE_hat - 1.96*SE_hat,
                 CI_h = ATE_hat + 1.96*SE_hat,
                 covered = CI_l <= 0.30 & 0.30 <= CI_h ) %>%
  group_by( method ) %>%
  summarise( coverage = mean( covered ) )
```

```
## # A tibble: 3 x 2
##   method coverage
##   <chr>     <dbl>
## 1 Agg       0.942
## 2 LR        0.908
## 3 MLM       0.943
```

Our coverage is all lower than expected, with linear regression being around 5 percentage points too low and the other two methods being about 2.5 percentage points low. Linear regression is taking a hit from the bias term.

7.7 Further measures of performance

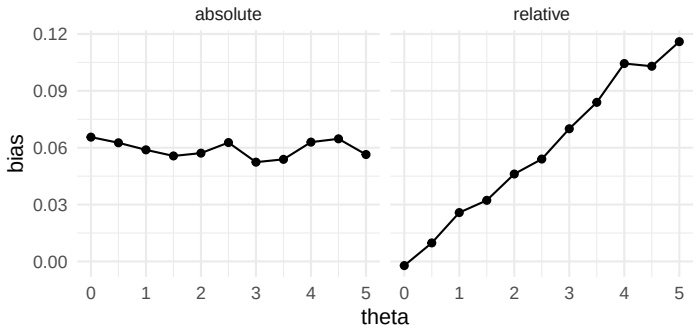
Depending on the model and estimation procedures being examined, a range of different criteria might be used to assess estimator performance. For point estimation, we have seen bias, variance and MSE as the three core measures of performance. Other criteria exist, such as the median bias and the median absolute deviation of T, where we use the median $\tilde{T}$ of our estimates rather than the mean $\bar{T}$. We can also calculate performance relative to the target parameter (like we saw for estimated standard errors), rather than assessing performance on an absolute scale as we saw above for point estimates.

7.7.1 Relative vs. Absolute Criteria

In the above, we presented the absolute criteria for the point estimators, where we obtain measures such as bias or uncertainty in the units on the scale of the point estimate (which is usually on the scale of some outcome of interest). By contrast, for parameters such as a standard error, that measure scale, or that are always strictly positive, it often makes sense to quantify performance using *relative* criteria. Relative criteria are very similar to the absolute criteria discussed for point estimators, but are defined as proportions of the target parameter, rather than as differences. While usually we use absolute criteria for point estimators and relative criteria for standard error estimators, this is not a fixed rule.

In general, we do not expect 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. We present two hypothetical scenarios in the figures below.



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 would be a better summary criterion.

Performance relative to a baseline estimator. More broadly, one often will 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. This could be critical to, for example, average simulations that have radically different sample sizes, where we would expect all estimators' performance measures to improve as sample size grows.

While a powerful tool, standardization 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 due to the behavior of the reference method in the denominator. These relative ratios can end up being confusing to interpret due to this tension.

7.7.2 Robust measures of performance

The usual bias, variance and MSE measures can be sensitive to outliers. If an estimator generally does well, except for an occasional large mistake, we would see poor overall performance by these classic measures. Instead, we might turn to quantities such as the median bias (sort all the estimation errors across the simulation scenarios, and take the middle), or the Median Absolute Distance (MAD, where you take the median of the absolute values of the errors) as a measure of performance.

Other robust measures are also possible, such as simply truncating all errors to a maximum size.

7.7.3 Summary of peformance measures

All the performance criteria we saw in this chapter 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	$\mathrm{E}\left[(T-\theta)^2\right]$	$\left(\bar{T}-\theta\right)^2 + S_T^2$
Relative bias	$\mathrm{E}(T)/\theta$	$\bar{T}/\theta$
Relative median bias	$\mathrm{M}(T)/\theta$	$\tilde{T}/\theta$
Relative MSE	$\mathrm{E}\left[(T-\theta)^2\right]/\theta^2$	$\frac{\left(\bar{T}-\theta\right)^2 + S_T^2}{\theta^2}$

- Bias and median bias are measures of whether the estimator is systemati-

cally 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. This is the true SE of the estimator.
- 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 outlier estimates. In general the RMSE can be driven up by a single bad egg. The MAD is less sensitive to this.

7.8 Uncertainty in 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 can calculate what fraction rejected the null for that 100. But due to random chance, we might see a higher, or lower, proportion rejected than what we would see if we ran the simulation forever.

To account for this estimation uncertainty we want 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 straightforward: we have an independent and identically distributed set of measurements, so statistical 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 for T	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	-
Power & Validity	$\sqrt{r_\alpha\left(1-r_\alpha\right)/R}$
Coverage	$\sqrt{\omega_\beta\left(1-\omega_\beta\right)/R}$
Expected length	$\sqrt{S_W^2/R}$

For relative quantities, simply divide the criterion by the reference level. E.g., for relative bias T/θ, the standard error would be

$$SE(\frac{T}{\theta}) = \frac{1}{\theta}SE(T) = \sqrt{\frac{S_T^2}{(R\theta^2)}}.$$

For square rooted quantities, such as the SE for the SE (square root of Variance) or the RMSE (square root of MSE) we can use the Delta method. The Delta method says, if we assume $X \sim N(\theta, V/n)$, that we can approximate the distribution of $g(X)$ as

$$g(X) \sim N\left(g(\theta),\ \ g'(\theta)^2 \cdot \frac{V}{n}\right),$$

where $g'(x)$ is the derivative of $g(x)$. In other words, $\widehat{SE}(g(X)) \approx \widehat{SE}(\theta)g'(\theta)$. For estimation, we can plug in $\hat{\theta}$ into the above. For the square root, we have $g(x) = \sqrt{(}x)$ and $g'(x) = 1/2\sqrt{(}x)$. This gives, for example

$$SE(\widehat{RMSE}) = \frac{1}{2\widehat{MSE}}\widehat{SE}(\widehat{MSE}).$$

7.8.1 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. The problem is we cannot use our clean expressions for MCSEs of relative performance measures since we are not taking the uncertainty of our denominator into account.

To properly assess the overall MCSE, we need to do something else. One approach 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:

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

This would be quite time-consuming to compute if we did it by brute force. However, a few algebra tricks provide 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!

7.8.2 Calculating MCSEs with the simhelpers package

The `simhelper` package is designed to calculate MCSEs (and the performance metrics themselves) for you. It is easy to use: take this set of simulation runs 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
##    <dbl> <dbl>     <dbl>    <dbl>  <dbl> <dbl>
##  1    50    50         0  0.0258  0.0954 0.934
##  2    50    50         0  0.00516 0.0848 0.986
##  3    50    50         0 -0.0798  0.0818 0.781
##  4    50    50         0 -0.0589  0.102  0.854
##  5    50    50         0  0.0251  0.118  0.942
```

```
##  6    50    50         0 -0.115   0.106  0.725
##  7    50    50         0  0.157   0.115  0.645
##  8    50    50         0 -0.213   0.121  0.543
##  9    50    50         0  0.509   0.117  0.139
## 10    50    50         0 -0.354   0.0774 0.206
## # i 7,990 more rows
## # i 2 more variables: lower_bound <dbl>,
## #   upper_bound <dbl>
```

We can calculate performance metrics across all the range of scenarios:

```
welch_sub = filter( welch, n1 == 50, n2 == 50, mean_diff==0 )
calc_rejection(welch_sub, "p_val")
```

```
## # A tibble: 1 x 3
##       K rej_rate rej_rate_mcse
##   <int>    <dbl>         <dbl>
## 1  1000    0.048       0.00676
```

```
calc_coverage(welch_sub,lower_bound, upper_bound, mean_diff)
```

```
## # A tibble: 1 x 5
##       K coverage coverage_mcse width width_mcse
##   <int>    <dbl>         <dbl> <dbl>      <dbl>
## 1  1000    0.952       0.00676  1.25    0.00338
```

Using `tidyverse` it is easy to process across scenarios (more on experimental design and multiple scenarios later):

```
welch %>% group_by(n1,n2,mean_diff) %>%
  do( calc_rejection( ., "p_val" ) )
```

```
## # A tibble: 8 x 6
## # Groups:   n1, n2, mean_diff [8]
##      n1    n2 mean_diff     K rej_rate
##   <dbl> <dbl>     <dbl> <int>    <dbl>
## 1    50    50       0    1000    0.048
## 2    50    50       0.5  1000    0.34
## 3    50    50       1    1000    0.876
## 4    50    50       2    1000    1
## 5    50    70       0    1000    0.027
## 6    50    70       0.5  1000    0.341
## 7    50    70       1    1000    0.904
## 8    50    70       2    1000    1
## # i 1 more variable: rej_rate_mcse <dbl>
```

7.8.3 Cluster RCT, continued: Do 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 trends.

7.9 Exercises

1. 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 8

Project: Cronbach Alpha

In this section we walk through a case study of Cronbach Alpha to give an extended "project," or series of exercises, that illustrate writing a complete simulation generated by the filling out of the code skeleton we get from `simhelpers`'s `create_skeleton()` package.

8.1 Background

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 $\Phi = \left[\phi_{ij}\right]_{i,j=1}^{p}$, where ϕ_{ij} is the covariance of item i and item j on the test, across all students taking the test. 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.

Our goal is to examine the properties of A when the set of P items is *not* multivariate 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

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

8.2 Getting started

First create the skeleton of our simulation. We will then walk through filling in all the pieces.

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

8.3 The data-generating function

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?
```

We need to create and test a function that takes model parameters (and sample sizes and such) as inputs, and produces a simulated dataset. 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:

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

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


library(mvtnorm)

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
}
```

Note how we translate the target α to ICC for our DGP; we will see this type of translation more later on.

We check our method first to see if we get the right kind of data:

```
small_sample <- r_mvt_items(n = 8, p = 3, alpha = 0.73, df = 5)
small_sample
```

```
##                A          B           C
## [1,]  0.3960864 -1.1852399  0.76961907
## [2,] -1.9233787 -0.2571960  0.16371549
## [3,]  1.6380163 -2.0359160 -1.06573793
## [4,]  0.5677678  0.6208439  0.08550315
## [5,] -0.6875976  1.0509309 -0.09511529
## [6,] -0.7444079  0.2696595  0.34109648
## [7,] -0.6036548  2.1193208  0.31642037
## [8,] -0.3798721 -0.6797846 -1.00087935
```

It looks like we have 8 observations of 3 items, as desired.

To check that the function is indeed simulating data following the intended distribution, let's next 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.

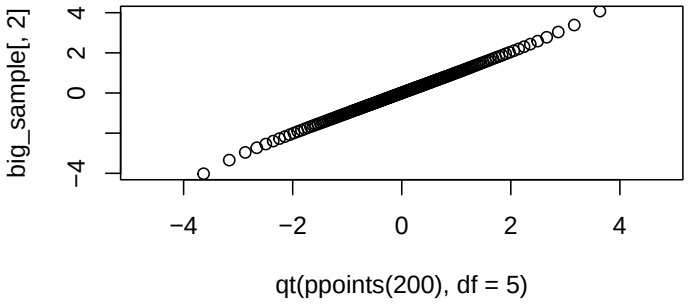
```
big_sample <- r_mvt_items(n = 100000, p = 4, alpha = 0.73, df = 5)
round(cor(big_sample), 3)
```

```
##       A     B     C     D
## A 1.000 0.409 0.412 0.410
## B 0.409 1.000 0.407 0.409
## C 0.412 0.407 1.000 0.405
## D 0.410 0.409 0.405 1.000
```

Is this what it should look like?

We can also check normality:

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



Looks good! A nice straight line.

8.4 The estimation function

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
```

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 on the right hand side 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 -0.8339665 1.261287
```

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

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

## Warning in alpha(x = small_sample): Some items were negatively correlated with the total scale
## should be reversed.
## To do this, run the function again with the 'check.keys=TRUE' option

## Some items ( A ) were negatively correlated with the total scale and
## probably should be reversed.
## To do this, run the function again with the 'check.keys=TRUE' option

## Warning in sqrt(Vtc): NaNs produced

##
## Reliability analysis
##  raw_alpha std.alpha G6(smc) average_r   S/N ase
##      -0.83     -0.61  -0.024     -0.15 -0.38 1.1
##    mean   sd median_r
##  -0.097 0.49    -0.37

## NULL
```

The next step is to evaluate these individual estimates and see how well our estimator A performs.

8.4.1 Exercices (Naive confidence intervals)

1. 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). Extend your simulation to calculate a confidence interval for each simulation round (put this code inside `estimate_alpha()`) and then calculate confidence interval coverage.

Your `estimate_alpha` would then give a result like this:

```
set.seed(40200)
dat = r_mvt_items(n = 50, p = 5, alpha = 0.9, df = 3 )
estimate_alpha(dat)
```

```
##           A          Var   CI_low   CI_high
## 1 0.9425904 0.0001647933 0.916916 0.9682647
```

2. You can calculate confidence intervals with coverage other than 95% by calculating an appropriate number of standard errors, z (usually just taken as 2, as above, for a nominal 95%), with

```
coverage = 0.95
z = qnorm( (1-coverage) / 2, lower.tail = FALSE )
z
```

```
## [1] 1.959964
```

Extend `estimate_alpha()` to allow for a specified coverage by adding a parameter, `coverage`, along with a default of 0.95. Revise the body of `estimate_alpha` to calculate a confidence interval with the specified coverage rate.

8.5 Estimator performance

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.

To start to get the code working that we would put into this function, it is useful to start with some simulation replicates to practice on. We can generate 1000 replicates using samples of size $n = 40$, $p = 6$ items, a true $\alpha = 0.8$, and $v = 5$ degrees of freedom:

```
one_run <- function( n, p, alpha, df ) {
    dat <- r_mvt_items(n = n, p = p, alpha = alpha, df = df)
    estimate_alpha(dat)
}
true_alpha = 0.7
results = rerun( 1000, one_run(40, 6, alpha=true_alpha, df=5) ) %>%
  bind_rows()
```

```
## Warning: `rerun()` was deprecated in purrr 1.0.0.
## i Please use `map()` instead.
##   # Previously
## rerun(1000, one_run(40, 6, alpha = true_alpha,
## df = 5))
##
##   # Now
## map(1:1000, ~ one_run(40, 6, alpha = true_alpha,
## df = 5))
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to
## see where this warning was generated.
```

8.5.1 Exercises (Calculating Performance)

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

1. With the parameters specified above, calculate the bias of A. Also calculate the Monte Carlo standard error (MCSE) of the bias estimate.

2. Estimate the true Standard Error of A.
3. Calculate the mean squared error of A.
4. Calculate the relative bias of the asymptotic variance estimator.
5. Using the work from above, wrap your code in an `alpha_performance()` function that takes the results of uses `run_alpha_sim` and returns a one-row data frame with columns corresponding to the bias, mean squared error, relative bias of the asymptotic variance estimator.

E.g.,

```
alpha_performance(results, true_alpha)
```

```
##          bias     bias_SE        SE       MSE
## 1 -0.02329445 0.003741931 0.1183302 0.1205433
##    bias_Var
## 1 0.5078144
```

6. Extend your function to add in the MCSEs for the SE and MSE. Code up the skewness and kurtosis values by hand, using the formula in the MCSE section of the performance measure chapter.
7. **(Challenge problem)** Code up a jackknife MCSE function to calculate the MCSE for the relative bias of the asymptotic variance estimator. Use the following template that takes a vector of point estimates and associated standard errors.

```
jackknife_MCSE <- function( estimates, SEs ) {
  # code
}
```

You would use this function as:

```
jackknife_MCSE( alpha_reps$A, sqrt( alpha_reps$Var ) )
```

8.6 Replication (and the simulation)

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 (this allows for reproducability, see Chapter **?**). 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 <- rerun(iterations, {
                dat <- dgm(model_params)
                estimate(dat, design_params)
              }) %>%
    bind_rows()


  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 (we pull out the code to replicate into its own method, `one_run()`, which helps with debugging):

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


one_run <- function( n, p, alpha, df ) {
    dat <- r_mvt_items(n = n, p = p, alpha = alpha, df = df)
    estimate_alpha(dat)
}


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

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

  results <-
    rerun(iterations, one_run(n, p, alpha, df) ) %>%
    bind_rows()

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

8.7 Extension: Confidence interval coverage

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 our transformed parameter and estimator be

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

and our transformed variance estimator be

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

(This expression comes from a Delta method expansion on A.)

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:

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

8.8 A taste of multiple scenarios

In the previous sections, we've created code that will generate a set of performance estimates, given a set of parameter values. We can 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)
```

```
## Warning: `rerun()` was deprecated in purrr 1.0.0.
## i Please use `map()` instead.
##   # Previously
##   rerun(100, one_run(n, p, alpha, df))
```

```
##
##   # Now
##   map(1:100, ~ one_run(n, p, alpha, df))
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to
## see where this warning was generated.
```

```
##          bias    bias_SE        SE       MSE
## 1 -0.01211522 0.01088426 0.1088426 0.1089725
##    bias_Var
## 1 0.4913622
```

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

```
##         bias    bias_SE        SE        MSE
## 1 -0.0117337 0.00761985 0.0761985 0.07671916
##    bias_Var
## 1 0.4727169
```

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

```
##          bias    bias_SE        SE       MSE
## 1 -0.02601087 0.01120983 0.1120983 0.1145292
##    bias_Var
## 1 0.4319068
```

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

```
##          bias     bias_SE         SE        MSE
## 1 0.007758062 0.006203414 0.06203414 0.06220884
##    bias_Var
## 1 0.5299059
```

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), but let's get a small taste of doing this now. In particular, the following code will 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)`) via `map_df()`:

```
alpha_true_seq <- seq(0.5, 0.9, 0.1)
results <- map_df( alpha_true_seq,
                   run_simulation,
                   R = 100,
                   n = 50, p = 5, df = 5 )
```

How does coverage change for different values of A?

8.8.1 Exercises

1. Show the inverse transform of $B = g(A)$ gives the above expression.
2. Make a new function, `estimate_alpha_xform()` that, given a dataset, calculates a confidence interval for α following the method described above.
3. Using the modified `estimate_alpha_xform()`, generate 1000 replicated confidence intervals for $n = 40$, $p = 6$ items, a true $\alpha = 0.8$, and $v = 5$ degrees of freedom. Using these replicates, calculate the true coverage rate of the confidence interval. Also calculate the Monte Carlo standard error (MCSE) of this coverage rate.
4. Calculate the average length of the confidence interval for α, along with its MCSE.
5. Compare the results of this approach to the more naive approach. Are there gains in performance?
6. *Challenge* Derive the variance expression for the transformed estimator using the Delta method on the variance expression for A coupled with the transform. The Delta method says that:

$$var(f(A)) \approx \frac{1}{f'(\alpha)}(A - \alpha)^2.$$

Chapter 9

Case study: Attrition in a simple randomized experiment

Missingness of the outcome variable is a common problem in randomized experiments conducted with human participants. For experiments where the focus is on estimating average causal effects, many different strategies for handling attrition have been proposed. Gerber and Green (2012) describe several strategies, including:

- Complete-case analysis, in which observations with missing outcome data are simply dropped from analysis;
- Regression estimation under the assumption that missingness is independent of potential outcomes, conditional on a set of pre-treatment covariates; and
- Monotone treatment response bounds, which provides bounds on the average treatment effect among participants who would provide outcome data under any treatment condition, under the assumption that that the effect of treatment on providing outcome data is strictly non-negative.

In this case study, we will develop a data-generating process that allows us to study these different estimation strategies. To make things interesting, we will examine a scenario in which the data include covariates that influence the probability of providing outcome data under treatment and under control, as well as being predictive of the outcome.

To formalize these notions, let us define the following variables:

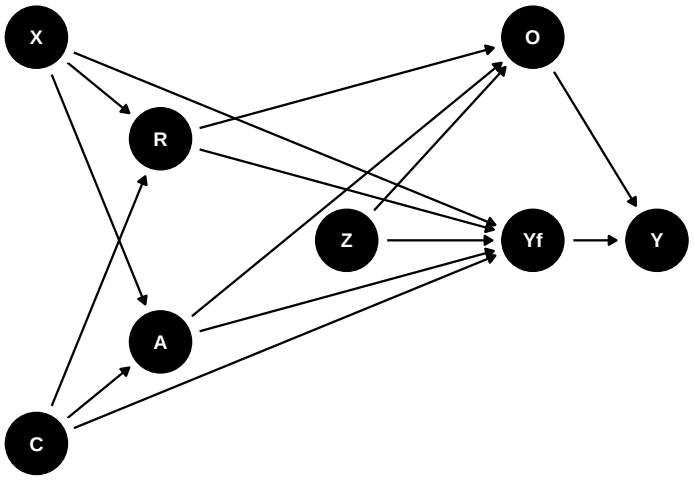
- X: a continuous covariate
- C: a binary covariate

- A: an indicator for whether a participant provides outcome data if assigned to treatment (latent)
- R: an indicator for whether a participant provides outcome data if assigned to control, given that $A = 1$ (latent)
- Z: a randomized treatment indicator
- Y^0, Y^1: potential outcomes (latent)
- Y^F: the outcome that would be observed if all participants were to respond (latent)
- Y: the measured outcome, which is only observed for some participants
- O: an indicator for whether the outcome Y is observed

We will posit that these variables are related as follows:

```
##
## Attaching package: 'ggdag'
```

```
## The following object is masked from 'package:stats':
##
##     filter
```



Now, let's lay out a more specific distributional model:

$$
\begin{aligned}
X &\sim N(0, 1) \\
C &\sim Bern(\kappa) - \kappa \\
Z &\sim Bern(0.5) \\
A &\sim Bern\left(\pi_A(C, X)\right) \\
R &\sim Bern\left(\pi_R(C, X)\right)
\end{aligned}
$$

where the response indicators follow the models

$$
\begin{aligned}
\text{logit } \pi_A(C, X) &= \alpha_{A0} + \alpha_{A1} C + \alpha_{A2} X \\
\text{logit } \pi_R(C, X) &= \alpha_{R0} + \alpha_{R1} C + \alpha_{R2} X.
\end{aligned}
$$