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.

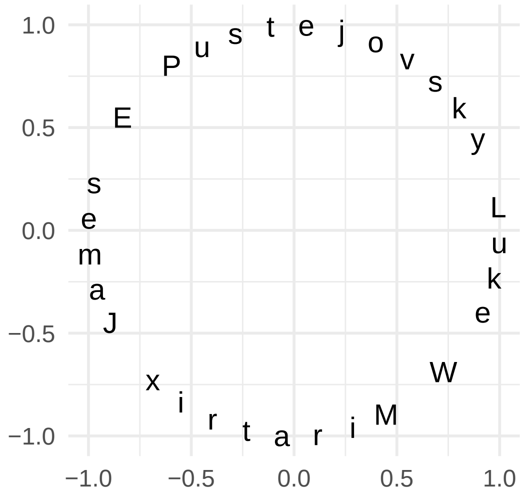
Monte Carlo simulations are an essential tool of inquiry for quantitative methodologists and students of statistics, useful both for small-scale or informal investigations and for formal methodological research. In this monograph, we aim to provide an introduction to the logic and mechanics of designing 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 (Miratrix et al., 2021)?

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 schols by order of preference. The central office then assigns students to schools via a lottery procedure where each student gets a lottery number that breaks ties when there are too many students desiring to go to a specific school. As a consequence, students have a random probability of assignment to the schools on their list, depending on their choices, the choices of other students, and their lottery numbers.

We can 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 (Abdulkadiroğlu et al., 2017).

For another example, one that possibly illustrates the perils of simulation as taking us away from results that pass face validity, Staiger and Rockoff (2010) 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

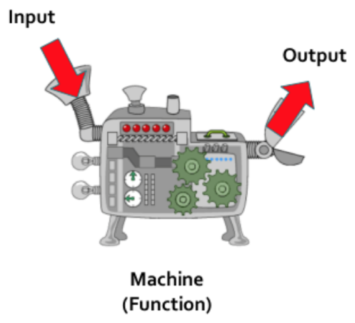


Figure 1.1: A function as a machine

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] 1.095781e-68
```

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

```
## [1] 0.2236271
```

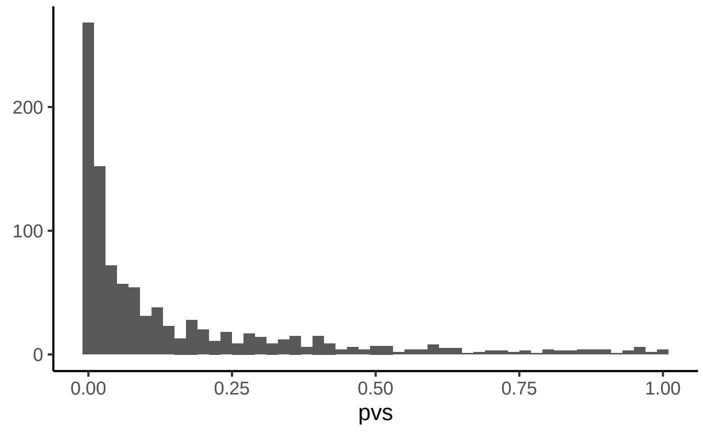
```
one_run( 10, 0.3 )
```

```
## [1] 0.6117171
```

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:

```r
# 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.

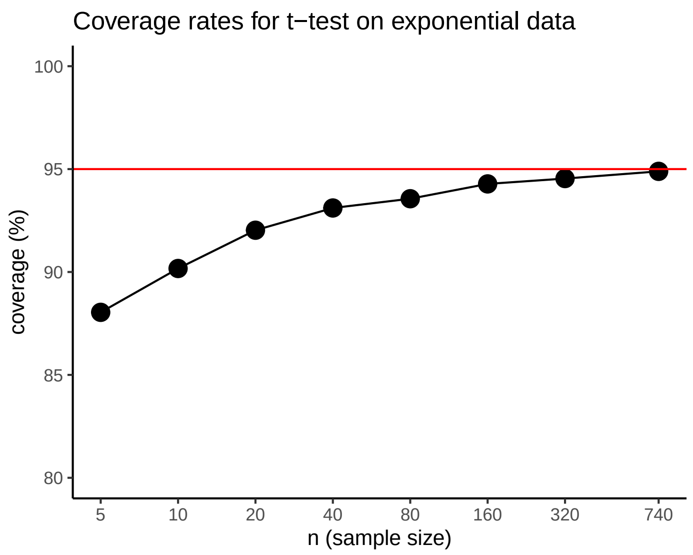
```
run.experiment = function( n ) {
  rps = replicate( 10000, {
    dat = rexp( n )
    tt = t.test( dat )
    findInterval( 1, tt$conf.int )
  })

  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:

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

```r
# 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.

```r
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:

```r
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:

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

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

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

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

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

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

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

```
## # A tibble: 15 x 2
##    group      x
##    <int>  <dbl>
##  1     1 -0.842
##  2     1  1.14
##  3     1  2.31
##  4     2  1.40
##  5     2  1.92
##  6     2  2.54
##  7     2 -1.09
##  8     2  4.47
##  9     2 -0.658
## 10     3  6.39
## 11     3  5.72
## 12     4  5.25
## 13     4  5.02
## 14     4  4.66
## 15     4  6.83
```

We have made a small dataset of group membership and outcome. We note that there are many different and legitimate ways of doing this in R. E.g., we could generate each group separately, and then stack our groups instead of using `rep` to do it all at once. In general, we advocate the adage that if you can do it at all, then you should feel good about yourself. 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.

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

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

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

```r
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.

```r
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 specification 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
```

```r
  # 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.

```r
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:

```r
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:

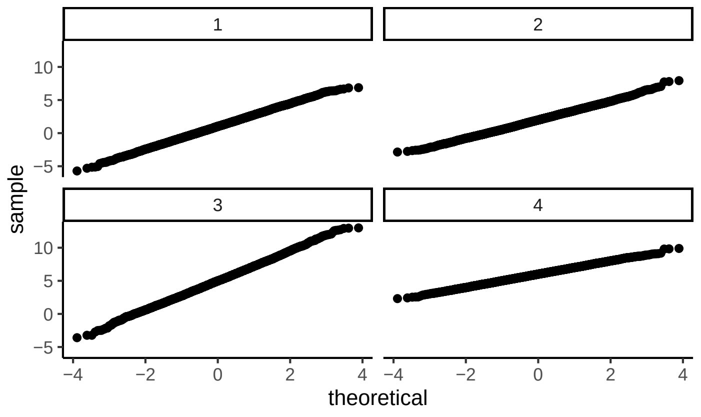
```
check_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                            sample_size = rep(10000, 4))

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 $j = 1, \dots, J$). 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 heterogeniety in treatment impact; this could make interpreting the results very confusing.

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

$$ICC = \frac{\sigma^2_u}{\sigma^2_\epsilon + \sigma^2_u}.$$

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 $\sigma^2_u + \sigma^2_\epsilon = 1$. It also gives us $ICC = \sigma^2_u$, and $\sigma^2_\epsilon = 1 - ICC$.

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^2_u 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:

```r
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
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:

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

Our three analysis functions are then Multilevel Regression (MLM):

```r
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):

```r
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):

```r
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.

```r
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).

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

  # fill in the guts here

  return(pval = pval)
}
```

```r
# 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
```

```r
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.

```r
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 Sundberg (2003).

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 Gerber and Green (2012). 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.

```r
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

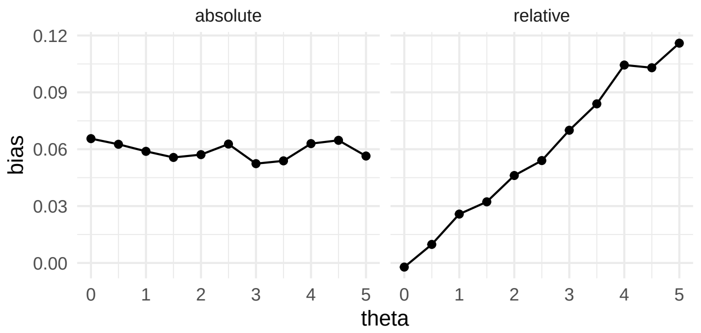
Depending on the model and estimation procedures being examined, a range of different criteria might be used to assess estimator performance. For point estimation, we have seen bias, variance and MSE as the three core measures of performance. Other criteria exist, such as the median bias and the median 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	$\sqrt{\frac{1}{R}\left[S_T^4(k_T-1)+4S_T^3g_T\left(\bar{T}-\theta\right)+4S_T^2\left(\bar{T}-\theta\right)^2$
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:

```r
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.

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

8.3 The data-generating function

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

```r
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.

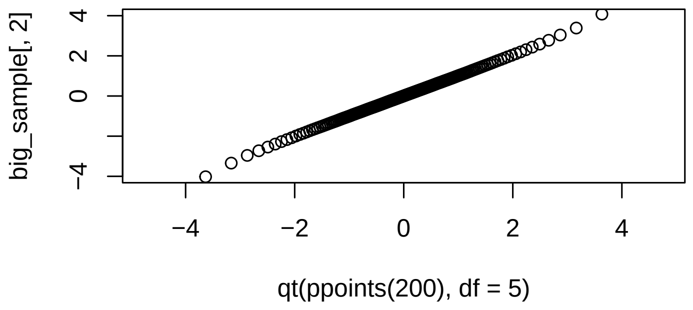
```r
big_sample <- r_mvt_items(n = 100000, p = 4, alpha = 0.73, df = 5)
round(cor(big_sample), 3)
```

```
##       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:

```r
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:

```r
#------------------------------------------------------
# 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:

```r
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:

```r
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:

```r
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

```r
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.

```r
#------------------------------------------------------
# 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:

```r
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

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

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:

```r
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

Designing the multifactor simulation experiment

So far, we've created code that will give us results for a single combination of parameter values. In practice, simulation studies typically examine a range of different values, including varying the level of the true parameter values and perhaps also varying sample sizes, to explore a range of different scenarios. We either want reassurance that our findings are general, or we want to understand what aspects of the context lead to our found results. A single simulation gives us no hint as to either of these questions. It is only by looking across a range of settings that we can fully understand trade-offs, general rules, and limits. Let's now look at the remaining piece of the simulation puzzle: the study's experimental design.

Simulation studies often take the form of **full factorial** designed experiments. In full factorials, each factor (a particular knob a researcher might turn to change the simulation conditions) is varied across multiple levels, and the design includes *every* possible combination of the levels of every factor. One way to represent such a design is as a list of factors and levels.

For example, for the Cronbach alpha simulation, we might want to vary:

- the sample size, with values of 50 or 100; and
- the number of items, with values of 4 or 8.
- the true value of alpha, with values ranging from 0.1 to 0.9;
- the degrees of freedom of the multivariate *t* distribution, with values of 5, 10, 20, or 100;

We first express the simulation parameters as a list of factors, each factor having a list of values to explore. We will then run a simulation for every possible combination of these values. We call this a $2 \times 2 \times 9 \times 4$ factorial design, where each element is the number of options for that factor. Here is code that generates all the scenarios we will run given the above design, storing these combinations in a data frame, `params`, that represents the full experimental design:

```r
design_factors <- list(
  n = c(50, 100),
  p = c(4, 8),
  alpha = seq(0.1, 0.9, 0.1),
  df = c(5, 10, 20, 100)
)

params <- cross_df(design_factors)
params
```

```
## # A tibble: 144 x 4
##         n     p alpha    df
##     <dbl> <dbl> <dbl> <dbl>
##  1     50     4   0.1     5
##  2    100     4   0.1     5
##  3     50     8   0.1     5
##  4    100     8   0.1     5
##  5     50     4   0.2     5
##  6    100     4   0.2     5
##  7     50     8   0.2     5
##  8    100     8   0.2     5
##  9     50     4   0.3     5
## 10    100     4   0.3     5
## # i 134 more rows
```

See what we get? The parameters we would pass to `run.experiment()` correspond to the columns of our dataset. We have a total of $2 \times 2 \times 9 \times 4 = 144$ rows, each row corresponding to a simulation scenario to explore. With multifactor experiments, it is easy to end up running a lot of experiments!

9.1 Choosing parameter combinations

How do we go about choosing parameter values to examine? Choosing which parameters to use is a central part of good simulation design because the primary limitation of simulation studies is always their *generalizability*. On the one hand, it's difficult to extrapolate findings from a simulation study beyond the set of simulation conditions that were examined. On the other hand, it's often difficult or impossible to examine the full space of all possible parameter values, except for very simple problems. Even in the Cronbach alpha simulation, we've got four factors, and the last three could each take an infinite number of different levels, in theory. How can we come up with a defensible set of levels to examine?

The choice of simulation conditions needs to be made in the context of the problem or model that you're studying, so it's a bit difficult to offer valid, decontextualized advice. We can provide a few observations all the same:

1. For research simulations, it often is important to be able to relate your findings to previous research. This suggests that you should select parameter levels to make this possible, such as by looking at sample sizes similar to those examined in previous studies. That said, previous simulation studies are not always perfect (actually, there's a lot of really crummy ones out there!), and so prior work should not geneally be your sole guide or justification.

2. Generally, it is better to err on the side of being more comprehensive. You learn more by looking at a broader range of conditions, and you can always boil down your results to a more limited set of conditions for purposes of presentation.

3. It is also important to explore breakdown points (e.g., what sample size is too small for a method to work?) rather than focusing only on conditions where a method might be expected to work well. Pushing the boundaries and identifying conditions where estimation methods break will help you to provide better guidance for how the methods should be used in practice.

An important point regarding (2) is that you can be more comprehensive and then have fewer replications per scenario. For example, say you were planning on doing 1000 simulations per scenario, but then you realize there is some new factor that you don't think matters, but that you believe other researchers will worry about. You could add in that factor, say with four levels, and then do 250 simulations per scenario. The total work remains the same.

When analyzing the final simulation you can then first verify you do not see trends along this new factor, and then marganalize out the factor in your summaries of results. Marginalizing out a factor (i.e., averaging your performance metrics across the additional factor) is a powerful technique of making a claim about how your methods work *on average* across a *range* of scenarios, rather than for a specific scenario.

Overall, you generally want to vary parameters that you believe matter, or that you think other people will believe matter. The first is so you can learn. The second is to build your case.

Once you have identified your parameters you then have to decide on the levels of the parameter you will include in the simulation. There are three strategies you might take:

1. Vary a parameter over its entire range (or nearly so).
2. Choose parameter levels to represent realistic practical range.
    - Empirical justification based on systematic reviews of applications
    - Or at least informal impressions of what's realistic in practice
3. Choose parameters to emulate one important application.

In the above (1) is the most general—but also the most computationally intensive. (2) will focus attention, ideally, on what is of practical relevance to a practitioner. (3) is usually coupled with a subsequent applied data analysis, and in this case the simulation is often used to enrich that analysis. For example, if the simulation shows the methods work for data with the given form of the target application, people may be more willing to believe the application's findings.

Regardless of how you select your primary parameters, you should also vary nuisance parameters (at least a little) to test sensitivity of results. While simulations will (generally) never be fully generalizable, you can certainly make them so they avoid the obvious things a critic might identify as an easy dismissal of your findings.

To recap, as you think about your parameter selection, always keep the following design principles and acknowledgements:

- The primary limitation of simulation studies is **generalizability**.
- Choose conditions that allow you to relate findings to previous work.
- Err towards being comprehensive.
  - The goal should be to build an understanding of the major moving parts.
  - Presentation of results can always be tailored to illustrate trends.
- Explore breakdown points (e.g., what sample size is too small for applying a given method?).

And fully expect to add and subtract from your set of parameters as you get your initial simulation results! No one ever runs just a single simulation.

9.1.1 Choosing parameters for the Clustered RCT

Extending our case study presented in Section **?** to a multifactor simulation, let's think about how to design our full experiment.

So far, we have only investigated a single scenario at a time, although our modular approach does make exploring a range of scenarios by re-calling our simulation function relatively straightforward. But how do our findings generalize? When are the different methods differently appropriate? To answer this, we need to extend to a multifactor simulation to *systematically* explore trends across contexts for our three estimators. We begin by identifying some questions we might have, given our preliminary results.

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

For precision, we also saw that Linear Regression has a higher standard error.

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

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

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

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

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

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

When designing the final factors, it is important to ensure those factors are isolated, in that changing one of them is not changing a host of other things that might impact performance. For example, in our case, if we simply added more cross site variation by directly increasing the random effects for the clusters, our total variation will increase. If we see that methods deteriorate, we then have a confound: is it the cross site variation causing the problem, or is it the total variation? We therefore want to vary site variation while controlling total variation; this is why we use the ICC knob discussed in the section on the data generation process.

We might thus end up with the following for our factors:

```r
crt_design_factors <- list(
  n_bar = c( 20, 80, 320 ),
  J = c( 5, 20, 80 ),
  ATE = c( 0.2 ),
  size_coef = c( 0, 0.2 ),
  ICC = c( 0, 0.2, 0.4, 0.6, 0.8 ),
  alpha = c( 0, 0.5, 0.8 )
)
```

9.2 Using pmap to run multifactor simulations

To run simulations across all of our factor combinations, we are going to use a very useful method in the `purrr` package called `pmap()`. `pmap()` marches down a set of lists, running a function on each p-tuple of elements, taking the i^{th} element from each list for iteration i, and passing them as parameters to the specified function. `pmap()` then returns the results of this sequence of function calls as a list of results.

Here is a small illustration:

```
my_function <- function( a, b, theta, scale ) {
    scale * (a + theta*(b-a))
}

args = list( a = 1:3,
             b = 5:7,
             theta = c(0.2, 0.3, 0.7) )
purrr::pmap_dbl(  args, my_function, scale = 10 )
```

```
## [1] 18 32 58
```

One important note is the variable names for the lists being iterated over must correspond exactly to function arguments of the called function. Extra parameters can be passed after the function name; these will be held constant, and passed to each function call. See how `scale` is the same for all calls.

As we see above, `pmap()` has variants such as `_dbl` or `_df` just like the `map()` and `map2()` methods. These variants will automatically stack or convert the list of things returned into a tidier collection (for `_dbl` it will convert to a vector of numbers, for `_df` it will stack the results to make a large dataframe, assuming each thing returned is a little dataframe).

So far, this is great, but it does not quite look like what we want: our factors are stored as a dataframe, not three lists. This is where R gets interesting: in R, the columns of a dataframe are stored as a list of vectors or lists (with each of the vectors or lists having the exact same length). This works beautifully with `pmap()`. Witness:

```
args[[2]]
```

```
## [1] 5 6 7
```

```
a_df = as.data.frame(args)
a_df
```

```
##   a b theta
## 1 1 5   0.2
## 2 2 6   0.3
## 3 3 7   0.7
```

```
a_df[[2]]
```

```
## [1] 5 6 7
```

```
purrr::pmap_dbl( a_df, my_function, scale = 10)
```

```
## [1] 18 32 58
```

We can pass `a_df` to `pmap`, and `pmap` takes it as a list of lists, and therefore does exactly what it did before.

All of this means `pmap()` can run a specified function on each row of a dataset. Continuing the Cronbach Alpha simulation from above, we would have the following:

```
params$iterations <- 500
sim_results <-  params %>%
  mutate(res = pmap(., run_alpha_sim ) )
```

We add a column to `params` to record the desired 500 replications per condition. The above code calls our `run_alpha_sim()` method for each row of our list of scenarios we want to explore. Even better, we are storing the results **as a new variable in the same dataset**.

```
sim_results
```

```
## # A tibble: 144 x 6
##        n     p alpha    df iterations res
##    <dbl> <dbl> <dbl> <dbl>      <dbl> <lgl>
##  1    50     4   0.1     5        500 NA
##  2   100     4   0.1     5        500 NA
##  3    50     8   0.1     5        500 NA
##  4   100     8   0.1     5        500 NA
##  5    50     4   0.2     5        500 NA
##  6   100     4   0.2     5        500 NA
##  7    50     8   0.2     5        500 NA
##  8   100     8   0.2     5        500 NA
##  9    50     4   0.3     5        500 NA
## 10   100     4   0.3     5        500 NA
## # i 134 more rows
```

We are creating a **list-column**, where each element in our list column is the little summary of our simulation results for that scenario. Here is the third scenario, for example:

```
sim_results$res[[3]]
```

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

We finally use `unnest()` to expand the `res` variable, replicating the values of the main variables once for each row in the nested dataset:

```
library(tidyr)
sim_results <- unnest(sim_results, cols = res) %>%
  dplyr::select( -iterations )
sim_results
```

```
## # A tibble: 144 x 5
##        n     p alpha    df res
##    <dbl> <dbl> <dbl> <dbl> <lgl>
##  1    50     4   0.1     5 NA
##  2   100     4   0.1     5 NA
##  3    50     8   0.1     5 NA
##  4   100     8   0.1     5 NA
##  5    50     4   0.2     5 NA
##  6   100     4   0.2     5 NA
##  7    50     8   0.2     5 NA
##  8   100     8   0.2     5 NA
##  9    50     4   0.3     5 NA
## 10   100     4   0.3     5 NA
## # i 134 more rows
```

We can put all of this together in a a tidy workflow as follows:

```
sim_results <-
  params %>%
  mutate(res = pmap(., .f = run_alpha_sim)) %>%
  unnest(cols = res)
```

If we wanted to use parallel processing (more on this later) we can also simply use the `simhelpers` package (the following code is auto-generated by the `create_skeleton()` method as well):

```
plan(multisession) # choose an appropriate plan from the future package
evaluate_by_row(params, run_alpha_sim)
```

We finally save our results using tidyverse's `write_csv()`; see "R for Data Science" textbook, 11.5. We can ensure we have a directory by making one via `dir.create()` (see Section **?** for more on files):

```
dir.create("results", showWarnings = FALSE )
write_csv( sim_results, file = "results/cronbach_results.csv" )
```

9.3 Ways of repeating oneself

We have three core elements in our simulation: - Generate data - Analyze data - Assess performance

In arranging these elements, we have a choice: do we compute performance measures for each simulation scenario as we go (inside) vs. computing after we get all of our individual results (outside)?

INSIDE (aggregate as you simulate): In this approach, illustrated above, we, for each scenario defined by a specific combination of factors, run our simulation for that scenario, assess the performance, and then return a nice summary table of how well our methods did. This is the most straightforward, given what we have done so far: we have a method to run a simulation for a scenario, and we simply run that method for a bunch of scenarios and collate.

After the `pmap()` call, we end up with a dataframe with all our simulations, one simulation context per row (or maybe bundles, one for each method), with our measured performance outcomes. This is ideally all we need to analyze.

We have less data to store, and it is easier to compartmentalize. On the cons side, we have no ability to add new performance measures on the fly.

Overall, this seems pretty good. That being said, sometimes we might want to use a lot of disk space and keep much more. In particular, each row of the above corresponds to the summary of a whole collection of individual runs. We might instead store all of these runs, which brings us to the other approach.

OUTSIDE (keep all simulation runs): In this approach we do not aggregate, but instead, for each scenario, return the entire set of individual estimates. The benefit of this is, given the raw estimates, you can dynamically add or change how you calculate performance measures. You do, however, end up with massive amounts of data to store and manipulate.

To move from inside to outside, we just take the summarizing step out of `run_alpha_sim()`. E.g.,:

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

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

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

  results
}
```

Each call to `run_alpha_sim_raw()` now gives one row per simulation trial. We replicate our simulation parameters for each row.

```
run_alpha_sim_raw( 4, 50, 6, 0.5, 3 )
```

```
##           A      Var_A       CI_L      CI_U
## 1 0.5402995 0.01014358 0.29374115 0.7007831
## 2 0.3867717 0.01805035 0.05786946 0.6008526
## 3 0.4617342 0.01390704 0.17303773 0.6496454
## 4 0.4810237 0.01292815 0.20267301 0.6622008
```

The primary advantage of this is we can then generate new performance measures, as they occur to us, later on. The disadvantage is this result file will be R times as many rows as the older file, which can get quite, quite large.

But disk space is cheap! Here we run the same experiment with our more complete storage. Note how the `pmap_df` stacks the multiple rows from each run, giving us everything nicely bundled up:

```
params$res <- params %>%
    pmap( run_alpha_sim_raw, iterations = 500 )
sim_results_full <- unnest( params,
                            cols = res )
write_csv( sim_results_full, "results/cronbach_results_full.csv" )
```

We end up with a lot more rows:

```
nrow( sim_results_full )
```

```
## [1] 72000
```

```
nrow( sim_results )
```

```
## [1] 144
```

Compare the file sizes: one is several k, the other is around 20 megabytes.

```
file.size("results/cronbach_results.csv") / 1024
```

```
## [1] 3.961914
```

```
file.size("results/cronbach_results_full.csv") / 1024
```

```
## [1] 7445.908
```

9.3.1 Getting raw results ready for analysis

If we generated raw results then we need to collapse them by experimental run before calculating performance measures so we can explore the trends across the experiments. We do this by grouping our data and calling `alpha_performance()` for each group:

```
results <- sim_results_full %>%
  nest_by( n, p, alpha, df, .key = "alpha_sims" )
results
```

```
## # A tibble: 144 x 5
## # Rowwise:  n, p, alpha, df
##         n     p alpha    df         alpha_sims
##     <dbl> <dbl> <dbl> <dbl> <list<tibble[,6]>>
##  1    50     4   0.1     5          [500 x 6]
##  2    50     4   0.1    10          [500 x 6]
##  3    50     4   0.1    20          [500 x 6]
##  4    50     4   0.1   100          [500 x 6]
##  5    50     4   0.2     5          [500 x 6]
##  6    50     4   0.2    10          [500 x 6]
##  7    50     4   0.2    20          [500 x 6]
##  8    50     4   0.2   100          [500 x 6]
##  9    50     4   0.3     5          [500 x 6]
## 10    50     4   0.3    10          [500 x 6]
## # i 134 more rows
```

```
results$performance = map2( results$alpha_sims,
                              results$alpha, alpha_performance )
results <- results %>%
  dplyr::select( -alpha_sims ) %>%
  unnest( cols="performance" )
results
```

```
## # A tibble: 576 x 7
## # Groups:   n, p, alpha, df [144]
##         n     p alpha    df criterion           est
##     <dbl> <dbl> <dbl> <dbl> <chr>             <dbl>
##  1    50     4   0.1     5 alpha bias      -0.103
##  2    50     4   0.1     5 alpha RMSE       0.388
##  3    50     4   0.1     5 V relative bias  0.436
##  4    50     4   0.1     5 coverage         0.826
##  5    50     4   0.1    10 alpha bias      -0.0527
##  6    50     4   0.1    10 alpha RMSE       0.263
##  7    50     4   0.1    10 V relative bias  0.780
##  8    50     4   0.1    10 coverage         0.908
##  9    50     4   0.1    20 alpha bias      -0.0190
## 10    50     4   0.1    20 alpha RMSE       0.221
## # i 566 more rows
## # i 1 more variable: MCSE <dbl>
```

```
# NOTE: This is HORRIBLE.  There has to be a better way.
```

Now, if we want to add a performance metric, we can simply change `alpha_performance` and recalculate, without running the time-intensive simulations. Being able to re-analyze your results is generally a far easier fix than running all the simulations again after changing the `run_alpha_sim()` method.

The results of summarizing during the simulation vs. after as we just did leads to essentially the same place, however, although our old results are in long format (with one row per simulation metric vs. the metrics being columns).

9.4 Running the Cluster RCT multifactor experiment

Running our cluster RCT simulation is the exact same code as we have used before. Simulations take awhile to run so we save them so we can analyze at our leisure. Because we are not exactly sure what performance metrics we want, we will save our individual results, and calculate performance metrics on the full data. I.e., we are storing the individual runs, not the analyzed results!

The code is as follows:

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

The seed is for reproducibility; we discuss this more later on.

We then group by our simulation factors and calculate all our performance metrics at once directly. For example, here is the code for calculating performance measures across our simulation for the cluster randomized experiments example:

```
res <- readRDS( file = "results/simulation_CRT.rds" )

sres <-
  res %>%
  group_by( n_bar, J, ATE, size_coef, ICC, alpha, method ) %>%
  summarise(
    bias = mean(ATE_hat - ATE),
    SE = sd( ATE_hat ),
    RMSE = sqrt( mean( (ATE_hat - ATE )^2 ) ),
    ESE_hat = sqrt( mean( SE_hat^2 ) ),
    SD_SE_hat = sqrt( sd( SE_hat^2 ) ),
    power = mean( p_value <= 0.05 ),
    R = n(),
    .groups = "drop"
  )
sres
```

```
## # A tibble: 810 x 14
##    n_bar     J   ATE size_coef   ICC alpha method
##    <dbl> <dbl> <dbl>     <dbl> <dbl> <dbl> <chr>
##  1    20     5   0.2         0   0     0   Agg
##  2    20     5   0.2         0   0     0   LR
##  3    20     5   0.2         0   0     0   MLM
##  4    20     5   0.2         0   0     0.5 Agg
##  5    20     5   0.2         0   0     0.5 LR
##  6    20     5   0.2         0   0     0.5 MLM
##  7    20     5   0.2         0   0     0.8 Agg
##  8    20     5   0.2         0   0     0.8 LR
##  9    20     5   0.2         0   0     0.8 MLM
## 10    20     5   0.2         0   0.2   0   Agg
## # i 800 more rows
## # i 7 more variables: bias <dbl>, SE <dbl>,
## #   RMSE <dbl>, ESE_hat <dbl>, SD_SE_hat <dbl>,
## #   power <dbl>, R <int>
```

9.4.1 Making analyze_data() quiet

If we run our simulation when there is little cluster variation, we start getting a lot of messages and warnings from our MLM estimator. For example, from a single call we get:

```
res <- analyze_data(dat)
```

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

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

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

   ...

    # Compile our results
    tibble(
      method = c( "MLM", "LR", "Agg" ),
      ATE_hat = c( est1, est2, est3 ),
      SE_hat = c( se1, se2, se3 ),
      p_value = c( pv1, pv2, pv3 ),
      message = c( message1, 0, 0 ),
      warning = c( warning1, 0, 0 )
    )
}
```

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

```
res <- analyze_data(dat)
res
```

```
## # A tibble: 3 x 6
##   method ATE_hat SE_hat p_value message warning
##   <chr>    <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
## 1 MLM    -0.376   1.44    0.842       0       0
## 2 LR     -0.0973  0.784   0.921       0       0
## 3 Agg    -0.440   0.856   0.698       0       0
```

See? No more warnings, but we see the message as a variable in our results.

Chapter 10

Analyzing the multifactor experiment

Once we have performance measures for all our simulation scenarios, how do we explore them? For our Cluster RCT simulation, we have 270 different simulation runs across our factors (with three rows per simulation run, one for each method). How can we visualize and understand trends across this complex domain?

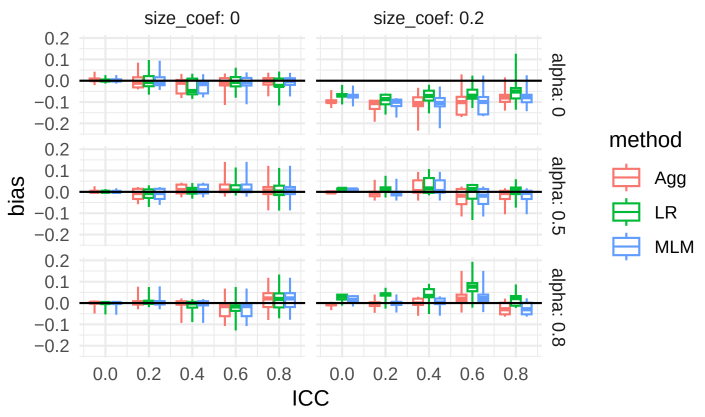
There are several techniques for summarizing across the data that one might use.

10.1 Bundling

As a first step, we might bundle the simulations by the primary factors of interest. We would then plot these bundles as box plots to see central tendency along with variation. With bundling, we would need a good number of simulation runs per scenario, so that the MCSE in the performance measures does not make our boxplots look substantially more variable than the truth.

For example, as a first step to understanding bias, we might bundle our results by ICC. In this code we are making groups of method by ICC level so we get side-by-side boxplots for each ICC level considered:

```r
res <- readRDS( "results/simulation_CRT.rds" )
ggplot( sres, aes( ICC, bias, col=method, group=paste0(ICC,method) ) ) +
  facet_grid( alpha ~ size_coef, labeller = label_both ) +
  geom_boxplot(coef = Inf) +
  geom_hline( yintercept = 0 ) +
  theme_minimal() +
  scale_x_continuous( breaks = unique( sres$ICC) )
```



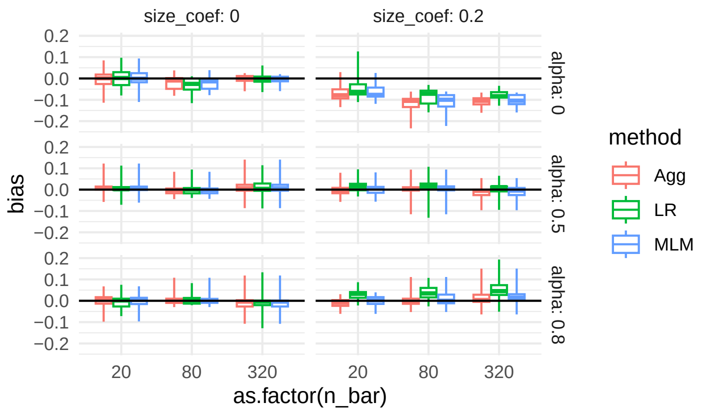
Each box is a collection of simulation trials. E.g., for `ICC = 0.6`, `size_coef = 0.2`, and `alpha = 0.8` we have 9 scenarios representing the varying level 1 and level 2 sample sizes:

```
filter( sres, ICC == 0.6, size_coef == 0.2,
        alpha == 0.8, method=="Agg" ) %>%
  dplyr::select( n_bar:alpha, bias )
```

```
## # A tibble: 9 x 7
##   n_bar     J   ATE size_coef   ICC alpha     bias
##   <dbl> <dbl> <dbl>     <dbl> <dbl> <dbl>    <dbl>
## 1    20     5   0.2       0.2   0.6   0.8  0.00799
## 2    20    20   0.2       0.2   0.6   0.8 -0.0445
## 3    20    80   0.2       0.2   0.6   0.8  0.0206
## 4    80     5   0.2       0.2   0.6   0.8  0.111
## 5    80    20   0.2       0.2   0.6   0.8 -0.0144
## 6    80    80   0.2       0.2   0.6   0.8  0.0132
## 7   320     5   0.2       0.2   0.6   0.8  0.151
## 8   320    20   0.2       0.2   0.6   0.8  0.0361
## 9   320    80   0.2       0.2   0.6   0.8  0.0394
```

We are seeing a few outliers for some of the boxplots, suggesting that there are other factors driving bias. We could try bundling along different aspects to see:

```
ggplot( sres, aes( as.factor(n_bar), bias, col=method, group=paste0(n_bar,method) ) )
  facet_grid( alpha ~  size_coef, labeller = label_both ) +
  geom_boxplot(coef = Inf) +
  geom_hline( yintercept = 0 ) +
  theme_minimal()
```



No progress there. Perhaps it is instability or MCSE. We make a note to investigate further, later on.

10.2 Aggregation

The boxplots are hard for seeing trends. Instead of bundling, we can therefore aggregate, to look at overall trends rather than individual simulation variation. This is especially important if the number of replicates within each scenario is small, because then each scenario's performance is measured with a lot of error.

With aggregation, we average over some of the factors, collapsing our simulation results down to fewer moving parts. This is better than having not had those factors in the first place! Averaging over a factor is a more general answer than having not varied the factor at all.

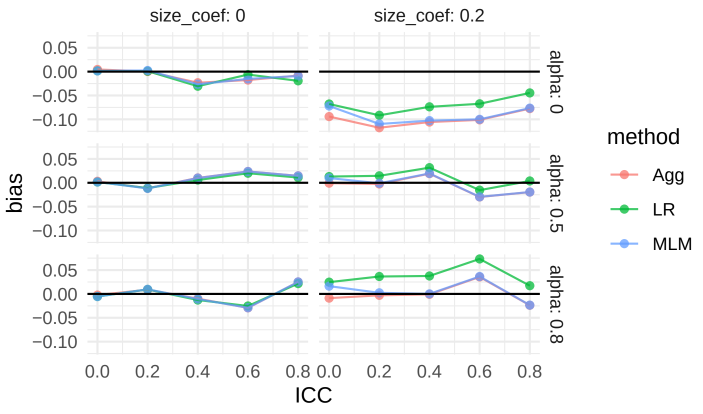
For example, if we average across ICC and site variation, and see how the methods change performance as a function of J, we would know that this is a general trend across a range of scenarios defined by different ICC and site variation levels. Our conclusions would then be more general than if we picked a single ICC and amount of site variation: in this latter case we would not know if we would see our trend more broadly.

Also, with aggregation, we can have a smaller number of replications per factor combination. The averaging will, in effect, give a lot more reps per aggregated performance measure.

A caution with aggregation is that it can be deceitful if you have scaling issues or extreme outliers. With bias, our scale is fairly well set, so we are good! But if we were aggregating standard errors over sample size, then the larger standard errors of the smaller sample size simulations (and the greater variability in estimating those standard errors) would swamp the standard errors of the larger sample sizes. Usually, with aggregation, we want to average over something we believe will not change massively over the marginalized-out factors. Alternatively, we can average over a relative measure, which tend to be more invariant and comparable across scenarios.

For our cluster RCT, we might aggregate as follows:

```
ssres <-
  sres %>%
  group_by( ICC, method, alpha, size_coef ) %>%
  summarise( bias = mean( bias ) )

ggplot( ssres, aes( ICC, bias, col=method ) ) +
  facet_grid( alpha ~  size_coef, labeller = label_both ) +
  geom_point( alpha=0.75 ) +
  geom_line( alpha=0.75 ) +
  geom_hline( yintercept = 0 ) +
  theme_minimal()
```



This shows that site variation leads to greater bias, but only if the coefficient for size is nonzero. We also see that all the estimators must be the same if site variation is 0, with the overplotted lines on the top row of the figure.

10.3 Regression Summarization

One can treat the simulation results as a dataset in its own right. In this case we can regress a performance measure against the methods and various factor levels to get “main effects” of how the different levels impact performance holding the other levels constant. The main effect of the method will tell us if a method is, on average, higher or lower than the baseline method. The main effect of the factors will tell us if that factor impacts the performance measure.

These regressions can also include interactions between method and factor, to see if some factors impact different methods differently. They can also include interactions between factors, which allows us to explore how the impact of a factor can matter more or less, depending on other aspects of the context.

For our cluster RCT, we might have, for example:

```
sres_f = sres %>%
  mutate( across( c( n_bar, J, size_coef, ICC, alpha ), factor ) )

M <- lm( bias ~ (n_bar + J + size_coef + ICC + alpha) * method,
         data = sres_f )
stargazer::stargazer(M, type = "text",
                     single.row = TRUE )
```

```
##
## ===================================================
##                              Dependent variable:
##                          ---------------------------
##                                     bias
## ---------------------------------------------------
## n_bar80                         -0.009 (0.007)
## n_bar320                        -0.006 (0.007)
## J20                              0.004 (0.007)
## J80                              0.006 (0.007)
## size_coef0.2                   -0.035*** (0.006)
## ICC0.2                          -0.004 (0.009)
## ICC0.4                          -0.002 (0.009)
## ICC0.6                          -0.003 (0.009)
## ICC0.8                           0.001 (0.009)
## alpha0.5                       0.055*** (0.007)
## alpha0.8                       0.053*** (0.007)
## methodLR                        -0.007 (0.014)
## methodMLM                        0.006 (0.014)
## n_bar80:methodLR                -0.001 (0.010)
## n_bar320:methodLR                0.001 (0.010)
## n_bar80:methodMLM               -0.001 (0.010)
## n_bar320:methodMLM              -0.002 (0.010)
## J20:methodLR                     0.006 (0.010)
## J80:methodLR                     0.005 (0.010)
## J20:methodMLM                    0.002 (0.010)
## J80:methodMLM                    0.002 (0.010)
## size_coef0.2:methodLR          0.030*** (0.008)
## size_coef0.2:methodMLM           0.006 (0.008)
## ICC0.2:methodLR                  0.003 (0.013)
## ICC0.4:methodLR                 0.0004 (0.013)
## ICC0.6:methodLR                  0.005 (0.013)
## ICC0.8:methodLR                  0.002 (0.013)
## ICC0.2:methodMLM                -0.006 (0.013)
## ICC0.4:methodMLM             -0.008 (0.013)
## ICC0.6:methodMLM             -0.008 (0.013)
## ICC0.8:methodMLM             -0.008 (0.013)
## alpha0.5:methodLR            -0.007 (0.010)
## alpha0.8:methodLR             0.004 (0.010)
## alpha0.5:methodMLM           -0.002 (0.010)
## alpha0.8:methodMLM           -0.0003 (0.010)
## Constant                    -0.034*** (0.010)
## ----------------------------------------------------
## Observations                       810
## R2                                0.302
## Adjusted R2                       0.270
## Residual Std. Error         0.046 (df = 774)
## F Statistic              9.547*** (df = 35; 774)
## ====================================================
## Note:                  *p<0.1; **p<0.05; ***p<0.01
```

We can quickly get a lot of features, and this approach can be hard to interpret. But picking out the significant coefficents does provide a lot of clues, rather rapidly. E.g., many features interact with the LR method for bias. The other methods seem less impacted.

10.4 Focus on subset, kick rest to supplement

Frequently researchers might simply filter the simulation results to a single factor level for some nuisance parameter. For example, we might examine ICC of 0.20 only, as this is a "reasonable" value given substance matter knowledge. We would then consider the other levels as a "sensitivity" analysis vaguely alluded to in our main report and placed elsewhere, such as an online supplemental appendix.

It would be our job, in this case, to verify that our reported findings on the main results indeed were echoed in our other, set-aside, simulation runs.

10.5 Analyzing results when some trials have failed

If methods fail, then this is something to investigate in its own right. Ideally, failure is not too common, so we can drop those trials, or keep them, without really impacting our overall results. But one should at least know what one is ignoring.

For example, in our cluster RCT, we know we have, at least sometimes, convergence issues. We know that ICC is an important feature, so we can explore how often we get a convergence message by ICC level:

```
res %>%
  group_by( method, ICC ) %>%
  summarise( message = mean( message ) ) %>%
  pivot_wider( names_from = "method", values_from="message" )
```

```
## Warning: There were 15 warnings in `summarise()`.
## The first warning was:
## i In argument: `message = mean(message)`.
## i In group 1: `method = "Agg"`, `ICC = 0`.
## Caused by warning in `mean.default()`:
## ! argument is not numeric or logical: returning NA
## i Run `dplyr::last_dplyr_warnings()` to see the
##   14 remaining warnings.
```

```
## # A tibble: 5 x 4
##     ICC   Agg    LR   MLM
##   <dbl> <dbl> <dbl> <dbl>
## 1   0      NA    NA    NA
## 2   0.2    NA    NA    NA
## 3   0.4    NA    NA    NA
## 4   0.6    NA    NA    NA
## 5   0.8    NA    NA    NA
```

We see that when the ICC is 0 we get a lot of convergence issues, but as soon as we pull away from 0 it drops off considerably. At this point we might decide to drop those runs with a message or keep them. In this case, we decide to keep. It shouldn't matter much in any case except the ICC = 0 case, and we know the convergence is due to trying to estimate a 0 variance, and thus is in some sense expected. Furthermore, we know people using these methods would likely ignore these messages, and thus we are faithfully capturing how these methods would be used in practice. We might eventually, however, want to do a separate analysis of the ICC = 0 context to see if the MLM approach actually falls apart, or if it is just throwing error messages.

10.6 A demonstration of visualization

Let's explore a case study comparing different visualizations of the same performance metric (in this case, power). The goal is to see how to examine a metric from several perspectives, and to see how to explore simulation results across scenarios.

For this example, we are going to look at a randomized experiment. We will generate control potential outcomes, and then add a treatment effect to the treated units. We assume the control group is normally distributed. We will generate a random data set, estimate the treatment effect by taking the difference in means and calculating the associated standard error, and generating a p-value using the normal approximation. (As we will see, this is not a good idea for small sample size since we should be using a t-test style approach.)

Violating our usual modular approach, we ae going to have a single function that does an entire step: it generates two groups of the given sizes, one treatment and one control, and then calculates the difference in means. It will then test this difference using the normal approximation.

The function also calculates and returns the true effect size of the DGP as the true treatment effect divided by the control standard deviation (useful for understanding power, shown later on).

```
run.one = function( nC, nT, sdC, tau, mu = 5, sdTau = 0 ) {
  Y0 = mu + rnorm( nC, sd=sdC )
  Y1 = mu + rnorm( nT, sd=sdC ) + tau + rnorm( nT, sd=sdTau )

  tau.hat = mean( Y1 ) - mean( Y0 )
  SE.hat = sqrt( var( Y0 ) / ( nC ) + var( Y1 ) / ( nT ) )

  z = tau.hat / SE.hat
  pv = 2 * (1 - pnorm( abs( z ) ))

  data.frame( tau.hat = tau.hat, SE.hat = SE.hat, z=z, p.value=pv )
}
```

Our function generates a data set, analyzes it, and give us back a variety of results as a one-row dataframe, as per usual:

```
run.one( nT=5, nC=10, sdC=1, tau=0.5 )
```

```
##     tau.hat    SE.hat          z   p.value
## 1 -0.246767 0.6607213 -0.3734812 0.7087903
```

In this case, our results are a mix of the parameters and estimated quantities.

We then write a function that runs our single trial multiple times and summarizes the results:

```
run.experiment = function( nC, nT, sdC, tau, mu = 5, sdTau = 0, R = 500 ) {

  eres = replicate( R, run.one( nC, nT, sdC, tau, sdTau=sdTau, mu=mu ),
                    simplify=FALSE )
  eres = bind_rows( eres )
  eres %>% summarise( E.tau.hat = mean( tau.hat ),
                      E.SE.hat = mean( SE.hat ),
                      power = mean( p.value <= 0.05 ) ) %>%
    mutate( nC=nC, nT=nT, sdC=sdC, tau=tau, mu=mu, sdTau=sdTau, R=R )
}
```

For performance, we have the average average treatment effect estimate `E.tau.hat`, the average Standard Error estimate `E.SE.hat`, and the power `power` (defined as the percent of time we reject at alpha=0.05, i.e., the percent of times our *p*-value was less than our 0.05 threshold):

Our function also adds in the details of the simulation (the parameters we passed to the `run.one()` call). This is an easy way to keep track of things.

We test our function to see what we get:

```
run.experiment( 10, 3, 1, 0.5, 5, 0.2 )
```

```
##    E.tau.hat  E.SE.hat power nC nT sdC tau mu
## 1  0.497077 0.6284977 0.218 10  3   1 0.5  5
##    sdTau   R
## 1    0.2 500
```

We next use the above to run a *multi-factor simulation experiment*. We are going to vary four factors: control group size, treatment group size, standard deviation of the units, and the treatment effect.

```
nC = c( 2, 4, 7, 10, 50, 500 )
nT = c( 2, 4, 7, 10, 50, 500 )
sdC = c( 1 )
tau = c( 0, 0.5, 1 )
sdTau = c( 0, 0.5 )

experiments = expand_grid( nC=nC, nT=nT, sdC=sdC, tau=tau, sdTau = sdTau )
experiments
```

```
## # A tibble: 216 x 5
##       nC    nT   sdC   tau sdTau
##    <dbl> <dbl> <dbl> <dbl> <dbl>
##  1     2     2     1   0     0
##  2     2     2     1   0     0.5
##  3     2     2     1   0.5   0
##  4     2     2     1   0.5   0.5
##  5     2     2     1   1     0
##  6     2     2     1   1     0.5
##  7     2     4     1   0     0
##  8     2     4     1   0     0.5
##  9     2     4     1   0.5   0
## 10     2     4     1   0.5   0.5
## # i 206 more rows
```

We next run an experiment for each row of our dataframe of experiment factor combinations, and save the results. Note our method of adding the design parametes into our results makes this step arguably moe clean than some of the other templates we have seen.

```
exp.res <- experiments %>% pmap_df( run.experiment, R=2000 )
dir.create("results", showWarnings = FALSE )
saveRDS( exp.res, file="results/Neyman_RCT_results.rds" )
```

The `R=500` after `run.experiment` passes the *same* parameter of $R = 500$ to each run (we run the same number of trials for each experiment). We can put it there rather than have it be a column in our list of factors to run.

Here is a peek at our results:

```
head( exp.res )
```

```
##      E.tau.hat  E.SE.hat  power nC nT sdC tau mu
## 1 -0.001525155 0.8818917 0.1975  2  2   1 0.0  5
## 2 -0.026286662 0.9307269 0.1915  2  2   1 0.0  5
## 3  0.485430674 0.8861287 0.2185  2  2   1 0.5  5
## 4  0.452375661 0.9363909 0.2095  2  2   1 0.5  5
## 5  1.029038283 0.8915091 0.3120  2  2   1 1.0  5
## 6  0.957434064 0.9347139 0.3025  2  2   1 1.0  5
##   sdTau    R
## 1   0.0 2000
## 2   0.5 2000
## 3   0.0 2000
## 4   0.5 2000
## 5   0.0 2000
## 6   0.5 2000
```

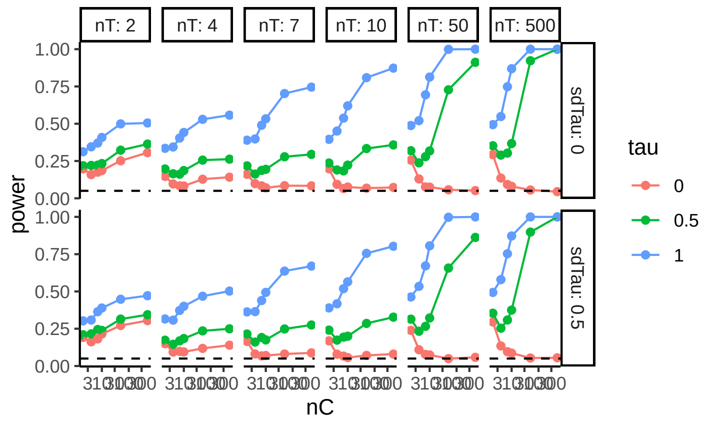
10.6.1 The initial analysis

We are ready to analyze.

We start with plotting. Plotting is always a good way to vizualize simulation results. We first make our tau into a factor, so `ggplot` behaves, and then plot all our experiments as two rows based on one factor (`sdTau`) with the columns being another (`nT`). (This style of plotting a bunch of small plots is called "many multiples" and is beloved by Tufte.) Within each plot we have the x-axis for one factor (`nC`) and multiple lines for the final factor (`tau`). The y-axis is our outcome of interest, power. We add a 0.05 line to show when we are rejecting at rates above our nominal α. This plot shows the relationship of 5 variables.

```
exp.res = exp.res %>% mutate( tau = as.factor( tau ) )
ggplot( exp.res, aes( x=nC, y=power, group=tau, col=tau ) ) +
  facet_grid( sdTau ~ nT, labeller=label_both ) +
  geom_point() + geom_line() +
```

```r
  scale_x_log10() +
  geom_hline( yintercept=0.05, col="black", lty=2)
```



We ae looking at power for different control and treatment group sizes. The tau is our treatment effect, and so for $\tau = 0$ we are looking at validity (false rejection of the null) and for the other τ power (noticing an effect when it is there). Notice that we are seeing elevated rejection rates under the null for small and even moderate sample sizes!

10.6.2 Focusing on validity

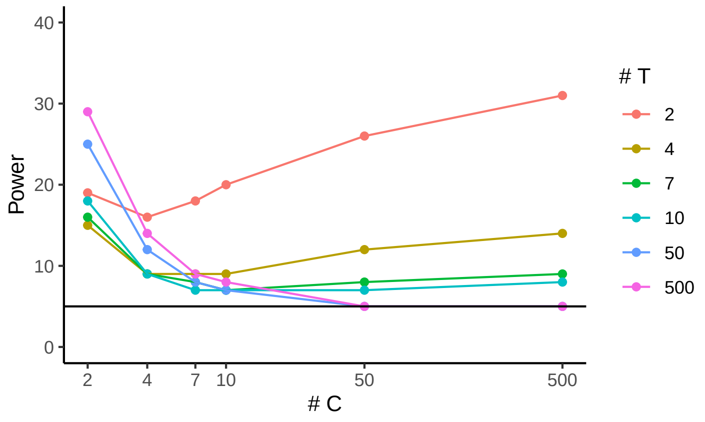
We can zoom in on specific simulations run, to get some more detail such as estimated power under the null for larger groups. Here we check and we are seeing rejection rates (`power`) of around 0.05, which is what we want.

```r
filter( exp.res, tau==0, nT >= 50, nC >= 50 ) %>%
  knitr::kable(digits=2)
```

E.tau.hat	E.SE.hat	power	nC	nT	sdC	tau	mu	sdTau	R
0	0.20	0.06	50	50	1	0	5	0.0	2000
0	0.21	0.05	50	50	1	0	5	0.5	2000
0	0.15	0.06	50	500	1	0	5	0.0	2000
0	0.15	0.05	50	500	1	0	5	0.5	2000
0	0.15	0.05	500	50	1	0	5	0.0	2000
0	0.16	0.06	500	50	1	0	5	0.5	2000
0	0.06	0.04	500	500	1	0	5	0.0	2000
0	0.07	0.06	500	500	1	0	5	0.5	2000

We can get fancy and look at rejection rate (power under `tau = 0`) as a function of both nC and nT using an interaction-style plot where we average over the other variables:

```
exp.res.rej <- exp.res %>% filter( tau == 0 ) %>%
  group_by( nC, nT ) %>%
  summarize( power = mean( power ) )

exp.res.rej = mutate( exp.res.rej, power = round( power * 100 ) )

ggplot( exp.res.rej, aes( x=nC, y=power, group=nT, col=as.factor(nT) ) ) +
  geom_point() + geom_line() +
  geom_hline( yintercept = 5 ) +
  scale_y_continuous( limits = c( 0, 40 ) ) +
  scale_x_log10( breaks = unique( exp.res.rej$nC ) ) +
  labs( x = "# C", y = "Power", colour = "# T" )
```



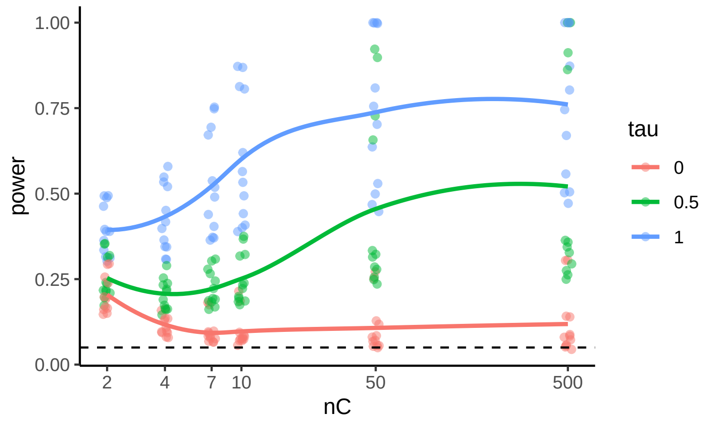
This plot focuses on the validity of our test. It shows that we have massively elevated rates when either the number of treated or control units is small (10 or below). It also shows that as the size of one group increases, if the other is small our rejection rates climb! Note how for 4 control units, the $n_T = 500$ line is above the others (except for the $n_T = 2$ line).

10.6.3 Looking at main effects

We can ignore all the other factors while we look at one specific factor of interest. This is looking at the **main effect** or **marginal effect** of the factor.

The easy way to do this is to let `ggplot` smooth our individual points on a plot. Be sure to also plot the individual points to see variation, however.

```
ggplot( exp.res, aes( x=nC, y=power, group=tau, col=tau ) ) +
  geom_jitter( width=0.02, height=0, alpha=0.5 ) +
  geom_smooth( se = FALSE ) +
  scale_x_log10( breaks=nC) +
  geom_hline( yintercept=0.05, col="black", lty=2)
```



Note how we see our individual runs that we marginalize over as the dots.

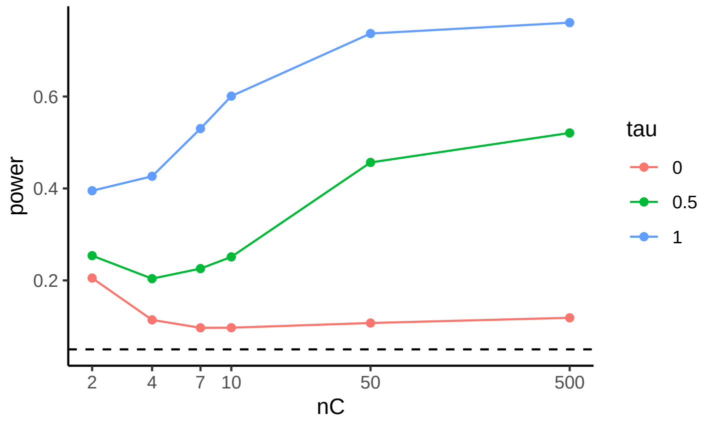
To look at our main effects we can also summarize our results, averaging our experimental runs across other factor levels. For example, in the code below we average over the different treatment group sizes and standard deviations, and plot the marginalized results.

To marginalize, we group by the things we want to keep. `summarise()` then averages over the things we want to get rid of.

```
exp.res.sum = exp.res %>% group_by( nC, tau ) %>%
  summarise( power = mean( power ) )
head( exp.res.sum )
```

```
## # A tibble: 6 x 3
## # Groups:   nC [2]
##      nC tau   power
##   <dbl> <fct> <dbl>
## 1     2 0     0.205
## 2     2 0.5   0.254
## 3     2 1     0.395
## 4     4 0     0.114
## 5     4 0.5   0.204
## 6     4 1     0.427
```

```
ggplot( exp.res.sum, aes( x=nC, y=power, group=tau, col=tau ) ) +
  geom_line() + geom_point() +
  scale_x_log10( breaks=nC) +
  geom_hline( yintercept=0.05, col="black", lty=2)
```

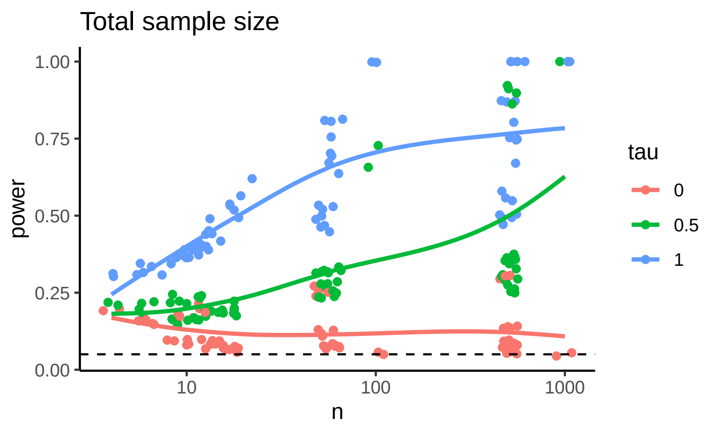


We can try to get clever and look at other aspects of our experimental runs. The above suggests that the smaller of the two groups is dictating things going awry, in terms of elevated rejection rates under the null. We can also look at things in terms of some other more easily interpretable parameter (here we switch to effect size instead of raw treatment effect).

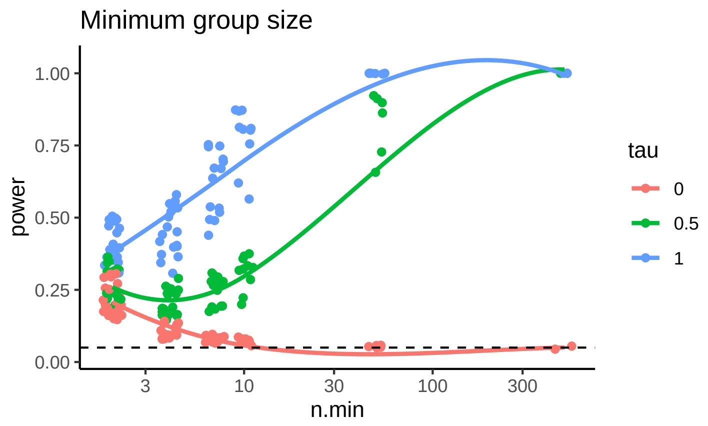
Given this, we might decide to look at total sample size or the smaller of the two groups sample size and make plots that way (we are also subsetting to just the `sd=1` cases as there is nothing really different about the two options; we probably should average across but this could reduce clarity of the presentation of results):

```
exp.res <- exp.res %>% mutate( n = nC + nT,
                               n.min = pmin( nC, nT ) )
```

```
ggplot( exp.res, aes( x=n, y=power, group=tau, col=tau ) ) +
  geom_jitter( width=0.05, height=0) +
  geom_smooth( se = FALSE, span = 1) +
  scale_x_log10() +
  geom_hline( yintercept=0.05, col="black", lty=2) +
  labs( title = "Total sample size" )
```



```r
ggplot( exp.res, aes( x=n.min, y=power, group=tau, col=tau ) ) +
  geom_jitter( width=0.05, height=0) +
  geom_smooth( se = FALSE, span = 1) +
  scale_x_log10() +
  geom_hline( yintercept=0.05, col="black", lty=2) +
  labs( title = "Minimum group size" )
```



Note the few observations out in the high `n.min` region for the second plot—this plot is a bit strange in that the different levels along the x-axis are assymetric with respect to each other. It is not balanced.

10.6.4 Recap

Overall, this exploration demonstrates the process of looking at a single performance metric (power) and refining a series of plots to get a sense of what the simulation is taking us. There are many different plots we might choose, and this depends on the messages we are trying to convey.

The key is to explore, and see what you can learn!

10.7 Exercises

1) For our cluster RCT, use the simulation results to assess how much better (or worse) the different methods are to each other in terms of confidence interval coverage. What scenarios tend to result in the worst coverage?

Chapter 11

Case study: Comparing different estimators

Features of this case study - Calculating performance metrics by estimator using tidyverse. - Visualization of simulation results. - Construction of the classic Bias + SE + RMSE performance plot.

In this case study we examine a simulation where we wish to compare different forms of estimator for estimating the same thing. We still generate data, evaluate it, and see how well our evaluation works. The difference is we now evaluate it multiple ways, storing how the different ways work.

For our simple working example we are going to compare estimation of the center of a symmetric distribution via mean, trimmed mean, and median (so the mean and median are the same). These are the three estimation strategies that we might be comparing in a paper (pretend we have "invented" the trimmed mean and want to demonstrate its utility).

We are, as usual, going to break building this simulation evaluation down into lots of functions to show the general framework. This framework can readily be extended to more complicated simulation studies. This case study illustrates how methodologists might compare different strategies for estimation, and is closest to what we might see in the "simulation" section of a stats paper.

11.1 The data generating process

For our data-generation function we will use the scaled t-distribution so the standard deviation will always be 1 but we will have different fatness of tails (high chance of outliers):

```r
gen.data = function( n, df0 ) {
    rt( n, df=df0 ) / sqrt( df0 / (df0-2) )
}
```

The variance of a t is $df/(df-2)$, so if we divide our observations by the square root of this, we will standardize them so they have unit variance. See, the standard deviation is 1 (up to random error, and as long as df0 > 2)!:

```r
sd( gen.data( 100000, df0 = 3 ) )
```

```
## [1] 1.01542
```

(Normally our data generation code would be a bit more fancy.)

We next define the parameter we want (in our case this is the mean, is what we are trying to estimate):

```r
mu = 0
```

11.2 The data analysis methods

We then write a function that takes data and uses all our different estimators on it. We return a data frame of the three estimates, with each row being one of our estimators. This is useful if our estimators return an estimate and a standard error, for example.

```r
analyze.data = function( data ) {
    mn = mean( data )
    md = median( data )
    mn.tr = mean( data, trim=0.1 )
    data.frame( estimator = c( "mean", "trim.mean", "median" ),
                estimate = c( mn, mn.tr, md ) )
}
```

Let's test:

```r
dt = gen.data( 100, 3 )
analyze.data( dt )
```

```
##   estimator     estimate
## 1      mean -0.044891002
## 2 trim.mean  0.005257327
## 3    median  0.004890847
```

Note that we have bundled our multiple methods into a single function. With complex methods we generally advocate a separate function for each method, but sometimes for a target simulation having a host of methods wrapped in a single function is clean and tidy code.

Also note the three lines of output for our returned value. This long-form output will make processing the simulation results easier. That being said, returning in wide format is also completely legitimate.

11.3 The simulation itself

To evaluate, do a bunch of times, and assess results. Let's start by looking at a specific case. We generate 1000 datasets of size 10, and estimate the center using our three different estimators.

```
raw.exps <- replicate( 1000, {
    dt = gen.data( n=10, df0=5 )
    analyze.data( dt )
}, simplify = FALSE )
raw.exps = bind_rows( raw.exps, .id = "runID" )
```

Note how our `.id` argument gives each simulation run an ID. This can be useful to see how the estimators covary.

We now have 1000 estimates for each of our estimators:

```
head( raw.exps )
```

```
##   runID estimator    estimate
## 1     1      mean -0.09919345
## 2     1 trim.mean -0.20887036
## 3     1    median -0.12237738
## 4     2      mean -0.19312165
## 5     2 trim.mean -0.22091715
## 6     2    median -0.18534152
```

11.4 Calculating performance measures for all our estimators

We then want to assess estimator performance for each estimator. We first write a function to calculate what we want from 1000 estimates:

```
estimator.quality = function( estimates, mu ) {
    RMSE = sqrt( mean( (estimates - mu)^2 ) )
    bias = mean( estimates - mu )
    SE = sd( estimates )
    data.frame( RMSE=RMSE, bias=bias, SE=SE )
}
```

The key is our function is estimation-method agnostic: we will use it for each of our three estimators. Here we evaluate our 'mean' estimator:

```
filter( raw.exps, estimator == "mean" ) %>%
    pull( estimate ) %>%
    estimator.quality( mu = mu )
```

```
##        RMSE        bias        SE
## 1 0.3318663 -0.01079814 0.3318566
```

Aside: Perhaps, code-wise, the above is piping having gone too far? If you don't like this style, you can do this:

```
estimator.quality( raw.exps$estimate[ raw.exps$estimator=="mean"], mu )
```

```
##        RMSE        bias        SE
## 1 0.3318663 -0.01079814 0.3318566
```

To do all our three estimators, we group by estimator and evaluate for each estimator. In tidyverse 1.0 `summarise` can handle multiple responses, but they will look a bit weird in our output, hence the 'unpack()' argument which makes each column its own column (if we do not unpack, we have a "data frame column" which is an odd thing).

```
raw.exps %>%
    group_by( estimator ) %>%
    summarise( qual = estimator.quality( estimate, mu = 0 ) ) %>%
    tidyr::unpack( cols=c(qual) )
```

```
## # A tibble: 3 x 4
##   estimator  RMSE     bias    SE
##   <chr>     <dbl>    <dbl> <dbl>
## 1 mean      0.332 -0.0108  0.332
## 2 median    0.331 -0.00855 0.331
## 3 trim.mean 0.311 -0.0105  0.311
```

We then pack up the above into a function, as usual. Our function takes our two parameters of sample size and degrees of freedom, and returns a data frame of results.

```
run.simulation = function( n, df0 ) {
    raw.exps <- replicate( 1000, {
        dt = gen.data( n=n, df0=df0 )
        analyze.data( dt )
    }, simplify = FALSE )
    raw.exps = bind_rows( raw.exps, .id = "runID" )

    rs <- raw.exps %>%
        group_by( estimator ) %>%
        summarise( qual = estimator.quality( estimate, mu = 0 ) ) %>%
        tidyr::unpack( cols=c( qual ) )

    rs
}
```

Our function will take our two parameters, run a simulation, and give us the results. We see here that none of our estimators are particularly biased and the trimmed mean has, possibly, the smallest RMSE, although it is a close call.

```
run.simulation( 10, 5 )
```

```
## # A tibble: 3 x 4
##   estimator  RMSE    bias    SE
##   <chr>     <dbl>   <dbl> <dbl>
## 1 mean      0.318 0.0136  0.318
## 2 median    0.315 0.00888 0.315
## 3 trim.mean 0.294 0.0111  0.294
```

Ok, now we want to see how sample size impacts our different estimators. If we also vary degrees of freedom we have a *three*-factor experiment, where one of the factors is our estimator itself. We are going to use a new clever trick. As before, we use `pmap()`, but now we store the entire dataframe of results we get back from our function in a new column of our original dataframe. See R for DS, Chapter 25.3. This trick works best if we have everything as a `tibble` which is basically a dataframe that prints a lot nicer and doesn't try to second-guess what you are up to all the time.

```
ns = c( 10, 50, 250, 1250 )
dfs = c( 3, 5, 15, 30 )
lvls = expand_grid( n=ns, df=dfs )

# So it stores our dataframe results in our lvls data properly.
lvls = as_tibble(lvls)

results <- lvls %>% mutate( results = pmap( lvls, run.simulation ) )
```

We have stored our results (a bunch of dataframes) in our main matrix of simulation runs.

```
print( results, n=4 )
```

```
## # A tibble: 16 x 3
##       n    df results
##   <dbl> <dbl> <list>
## 1    10     3 <tibble [3 x 4]>
## 2    10     5 <tibble [3 x 4]>
## 3    10    15 <tibble [3 x 4]>
## 4    10    30 <tibble [3 x 4]>
## # i 12 more rows
```

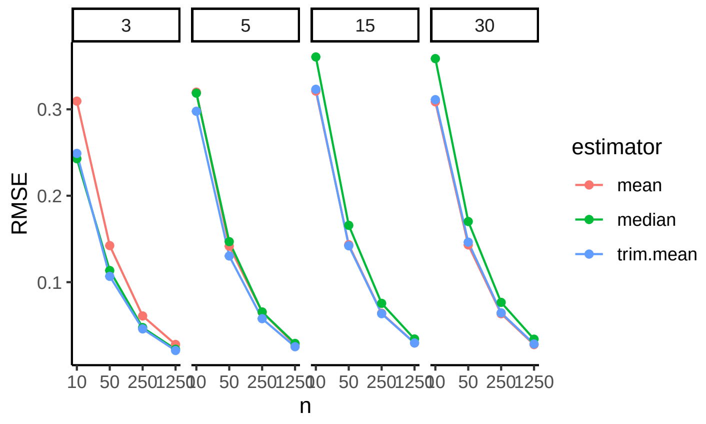
The `unnest()` function will stack up our dataframes, replicating the other columns in the main dataframe so it makes a nice rectangular dataset, all nice like. See (hard to read) R for DS Chapter 25.4.

```
results <- unnest( results, cols="results" )
results
```

```
## # A tibble: 48 x 6
##         n    df estimator  RMSE     bias    SE
##     <dbl> <dbl> <chr>     <dbl>    <dbl> <dbl>
##  1     10     3 mean      0.309 -0.0119  0.309
##  2     10     3 median    0.243 -0.0102  0.243
##  3     10     3 trim.mean 0.249 -0.0120  0.249
##  4     10     5 mean      0.320  0.00627 0.320
##  5     10     5 median    0.319  0.0121  0.319
##  6     10     5 trim.mean 0.298  0.00692 0.298
##  7     10    15 mean      0.321  0.00444 0.321
##  8     10    15 median    0.361  0.00925 0.361
##  9     10    15 trim.mean 0.323  0.00801 0.323
## 10     10    30 mean      0.309 -0.00229 0.309
## # i 38 more rows
```

And plot:

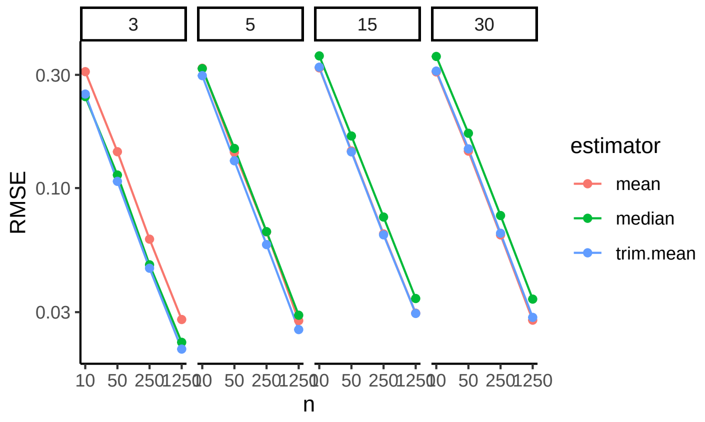
```
ggplot( results, aes(x=n, y=RMSE, col=estimator) ) +
            facet_wrap( ~ df, nrow=1 ) +
            geom_line() + geom_point() +
            scale_x_log10( breaks=ns )
```



11.5 Improving the visualization of the results

The above doesn't show differences clearly because all the RMSE goes to zero. It helps to log our outcome, or otherwise rescale. The logging version shows differences are relatively constant given changing sample size.

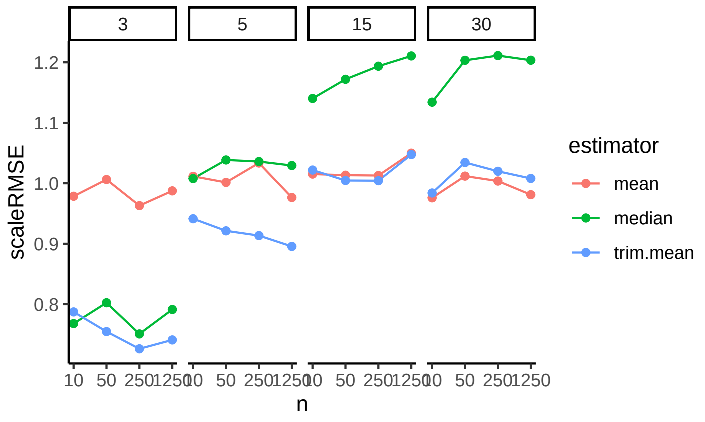
```r
ggplot( results, aes(x=n, y=RMSE, col=estimator) ) +
    facet_wrap( ~ df, nrow=1 ) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns ) +
    scale_y_log10()
```



Better is to rescale using our knowledge of standard errors. If we scale by the square root of sample size, we should get horizontal lines. We now clearly see the trends.

```r
results <- mutate( results, scaleRMSE = RMSE * sqrt(n) )
```

```r
ggplot( results, aes(x=n, y=scaleRMSE, col=estimator) ) +
    facet_wrap( ~ df, nrow=1) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns )
```



Overall, we see the scaled error of the mean it is stable across the different distributions. The trimmed mean is a real advantage when the degrees of freedom are small. We are cropping outliers that destabilize our estimate which leads to great wins. As the distribution grows more normal, this is no longer an advantage and we get closer to the mean in terms of performance. Here we are penalized slightly bye having dropped 10% of our data, so the standard errors will be slightly larger.

The median is not able to take advantage of the nuances of a data set because it is entirely determined by the middle value. When outliers cause real concern, this cost is minimal. When outliers are not a concern, the median is just worse.

Overall, the trimmed mean seems an excellent choice: in the presence of outliers it is far more stable than the mean, and when there are no outliers the cost of using it is small.

In terms of thinking about designing simulation studies, we see clear visual displays of simulation results can tell very clear stories. Eschew complicated tables with lots of numbers.

11.6 Extension: The Bias-variance tradeoff

We can use the above simulation to examine these same estimators when we the median is not the same as the mean. Say we want the mean of a distribution, but have systematic outliers. If we just use the median, or trimmed mean, we might have bias if the outliers tend to be on one side or another. For example, consider the exponential distribution:

```r
nums = rexp( 100000 )
mean( nums )
```

```
## [1] 0.9990221
```

```
mean( nums, trim=0.1 )
```

```
## [1] 0.83185
```

```
median( nums )
```

```
## [1] 0.6940772
```

Our trimming, etc., is *biased* if we think of our goal as estimating the mean. But if the trimmed estimators are much more stable, we might still wish to use them. Let's find out.

Let's generate a mixture distribution, just for fun. It will have a nice normal base with some extreme outliers. We will make sure the overall mean, including the outliers, is always 1, however. (So our target, μ is now 1, not 0.)

```
gen.data.outliers = function( n, prob.outlier = 0.05 ) {
    nN = rbinom( 1, n, prob.outlier )
    nrm = rnorm( n - nN, mean=0.5, sd=1 )
    outmean = (1 - (1-prob.outlier)/2) / prob.outlier
    outs = rnorm( nN, mean=outmean, sd=10 )
    c( nrm, outs )
}
```

Let's look at our distribution

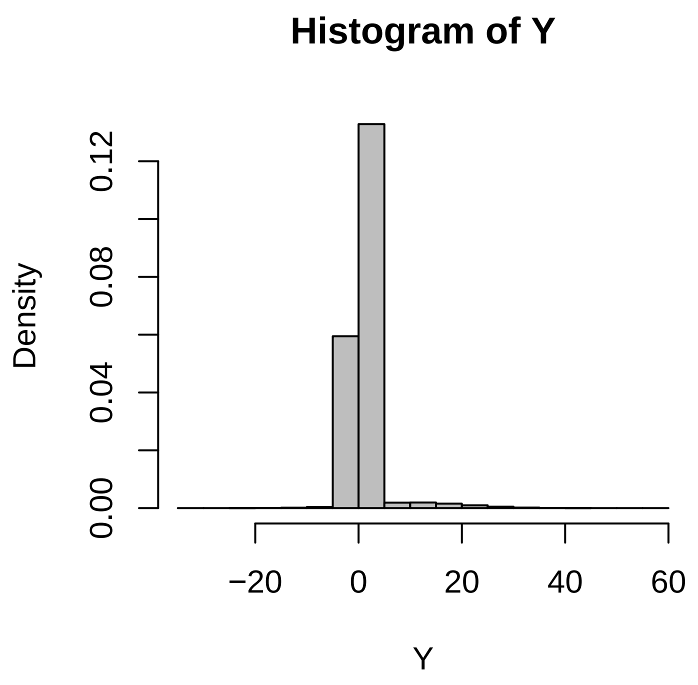
```
Y = gen.data.outliers( 10000000, prob.outlier = 0.05 )
mean( Y )
```

```
## [1] 1.000219
```

```
sd( Y )
```

```
## [1] 3.271463
```

```
hist( Y, breaks=30, col="grey", prob=TRUE )
```



We steal the code from above, modifying it slightly for our new function and changing our target parameter from 0 to 1:

```
run.simulation.exp = function( n ) {
    raw.exps <- replicate( 1000, {
        dt = gen.data.outliers( n=n )
        analyze.data( dt )
    }, simplify = FALSE )
    raw.exps = bind_rows( raw.exps, .id = "runID" )

    rs <- raw.exps %>%
        group_by( estimator ) %>%
        summarise( qual = estimator.quality( estimate, mu = 1 ) ) %>%
        tidyr::unpack( cols = c( qual ) )

    rs
}

res = run.simulation.exp( 100 )
res
```

```
## # A tibble: 3 x 4
##   estimator  RMSE     bias    SE
##   <chr>     <dbl>    <dbl> <dbl>
## 1 mean      0.315  0.00185 0.315
## 2 median    0.467 -0.450   0.127
## 3 trim.mean 0.450 -0.436   0.110
```

And for our experiment we vary the sample size

```
ns = c( 10, 20, 40, 80, 160, 320 )
lvls = tibble( n=ns )
```

```
results <- lvls %>%
  mutate( results = pmap( lvls, run.simulation.exp ) ) %>%
  unnest( cols = c(results) )
head( results )
```

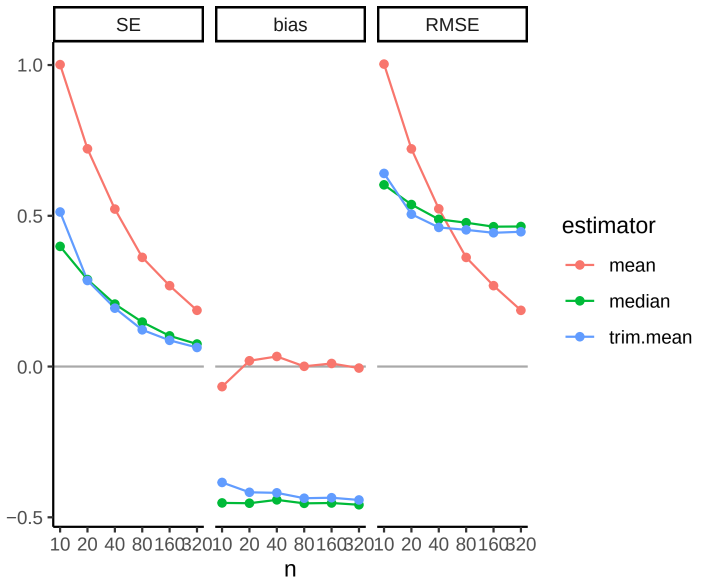
```
## # A tibble: 6 x 5
##       n estimator  RMSE    bias    SE
##   <dbl> <chr>     <dbl>   <dbl> <dbl>
## 1    10 mean      1.00  -0.0667 1.00
## 2    10 median    0.603 -0.452  0.399
## 3    10 trim.mean 0.641 -0.384  0.513
## 4    20 mean      0.722  0.0192 0.722
## 5    20 median    0.537 -0.453  0.289
## 6    20 trim.mean 0.505 -0.417  0.285
```

Here we are going to plot multiple outcomes. Often with the simulation study we are interested in different measures of performance. For us, we want to know the standard error, bias, and overall error (RMSE). To plot this we first gather our outcomes to make a long form dataframe of results:

```
res2 = gather( results, RMSE, bias, SE, key="Measure",value="value" )
res2 = mutate( res2, Measure = factor( Measure, levels=c("SE","bias","RMSE" )))
```

And then we plot, making a facet for each outcome of interest:

```
ggplot( res2, aes(x=n, y=value, col=estimator) ) +
    facet_grid( . ~ Measure ) +
    geom_hline( yintercept=0, col="darkgrey" ) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns ) +
    labs( y="" )
```



We see how different estimators have different biases and different uncertainties. The bias is negative for our trimmed estimators because we are losing the big outliers above and so getting answers that are too low.

The RMSE captures the trade-off in terms of what estimator gives the lowest overall *error*. For this distribution, the mean wins as the sample size increases because the bias basically stays the same and the SE drops. But for smaller samples the trimming is superior. The median (essentially trimming 50% above and below) is overkill and has too much negative bias.

From a simulation study point of view, notice how we are looking at three different qualities of our estimators. Some people really care about bias, some care about RMSE. By presenting all results we are transparent about how the different estimators operate.

Next steps would be to also examine the associated estimated standard errors for the estimators, seeing if these estimates of estimator uncertainty are good or poor. This leads to investigation of coverage rates and similar.

Chapter 12

Presentation of simulation results

Last chapter, we started to investigate how to present a multifactor experiment. In this chapter, we talk about some principles behind the choices one might make in generating final reports of a simulation. There are three primary approaches to the analysis and presentation of simulation results:

1. Tabulation
2. Visualization
3. Modeling

There are generally two primary goals for your results:

- Understand the effects of all of the factors manipulated in the simulation.
- Develop evidence that addresses your research questions.

For your final write-up, you will not want to present everything. A wall of numbers and observations will serve to pummel the reader, rather than inform them; readers rarely enjoy being pummeled, and the solution is quite often to skim such material while feeling hurt and betayed. Instead, you should present selected results that clearly illustrate the main findings from the study and anything unusual/anomolous. This will typically be with a few well-chosen figues. Then, in the text of your write-up, you might include examples that make specific numerical comparisons. Do not include too many of theses, and be sure to say why the numerical comparisons you include are important. Finally, have supplementary materials that contain further detail such as additional figures and analysis, and complete simulation results.

If you want to be a moral person worthy of the awards of Heaven, you should also provide reproducible code so others could, if so desired, rerun the simulation and conduct the analysis themselves. This last part provides a great legitimacy bump to your work: even if no one touches your code, knowing that they could builds confidence. People naturally think, “if that researcher is so willing to let me see what they actually did, then they must be fairly confident it does not contain too many horrendous mistakes and it is probably right.”

We briefly walk through the three modes of engaging with one’s simulation results, with a few examples taken from the literature.

12.1 Tabulation

Traditionally, simulation study results are presented in big tables. We think this doesn’t really make the take-aways of a simulation readily apparent. Perhaps tables are fine if… - they involve only a few numbers, and a few targeted comparisons - it is important to report *exact* values for some quantities

Unfortunately, simulations usually produce lots of numbers, and involve making lots of comparisons. You are going to want to show, for example, the relative performance of alternative estimators, or the performance of your estimators under different conditions for the data-generating model. This means a lot of rows, and a lot of dimensions. Tables can do two dimensions; when you try to cram more than that into a table, no one is particularly well served.

Furthermore, in simulation, exact values for your bias/RMSE/type-I error, or whatever, are not usually of interest. And in fact, we rarely have them due to Monte Carlo simulation error. The tables provide a false sense of security, unless you include uncertainty, which clutters your table even further.

Tables and simulations do not particularly well mix. In particular, if you are ever tempted into putting your table in landscape mode to get it to fit on the page, think again. It is often more useful and insightful to present results in graphs (Gelman, Pasarica, & Dodhia, 2002).

So, onwards.

12.2 Visualization

Visualization should nearly always be the first step in analyzing simulation results.

This often requires creating a *BUNCH* of graphs to look at different aspects of the data.

Helpful tools/concepts:

- Boxplots are often useful for depicting range and central tendency across many combinations of parameter values.

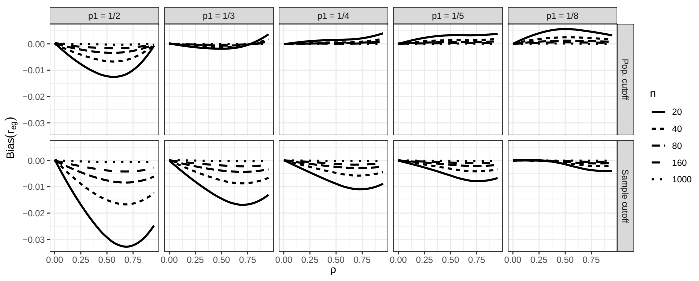
- Use color, shape, and line type to encode different factors
- Small multiples (faceting) can then encode further factors (e.g., varying sample size)

We next present a series of visualizations taken from our published work, illustrating some different themes behind visualization that we believe are important.

12.2.1 Example 1: Biserial correlation estimation

Our first example shows the bias of a biserial correlation estimate from an extreme groups design. This simulation was a $96 \times 2 \times 5 \times 5$ factorial design (true correlation for a range of values, cut-off type, cut-off percentile, and sample size). The correlation, with 96 levels, forms the x-axis, giving us nice performance curves. We use line type for the sample size, allowing us to easily see how bias collapses as sample size increases. Finally, the facet grid gives our final factors of cut-off type and cut-off percentile. All our factors, and nearly 5000 explored simulation scenarios, are visible in a single plot.

```
## `geom_smooth()` using formula = 'y ~ x'
```



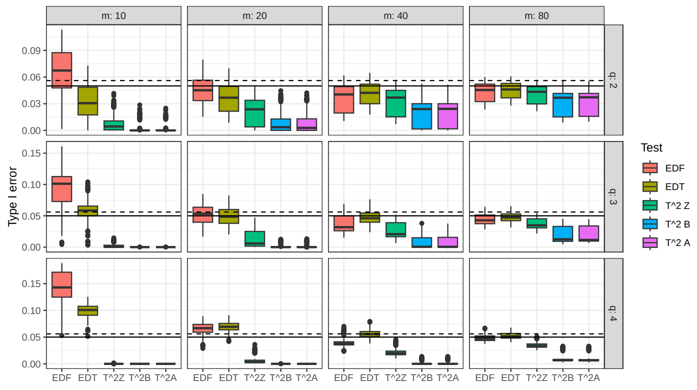
Source: Pustejovsky, J. E. (2014). Converting from d to r to z when the design uses extreme groups, dichotomization, or experimental control. Psychological Methods, 19(1), 92-112.

Note that in our figure, we have smoothed the lines with respect to `rho` using `geom_smooth()`. This is a nice tool for taking some of the simulation jitter out of an analysis to show overall trends more directly.

12.2.2 Example 2: Variance estimation and Meta-regression

- Type-I error rates of small-sample corrected F-tests based on cluster-robust variance estimation in meta-regression
- Comparison of 5 different small-sample corrections
- Complex experimental design, varying

– sample size (m)
– dimension of hypothesis (q)
– covariates tested
– degree of model mis-specification

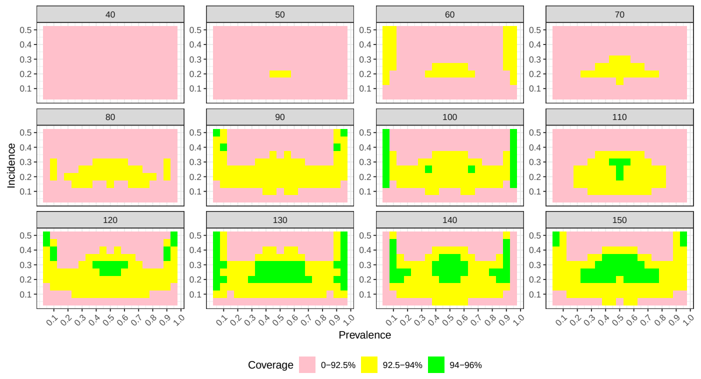


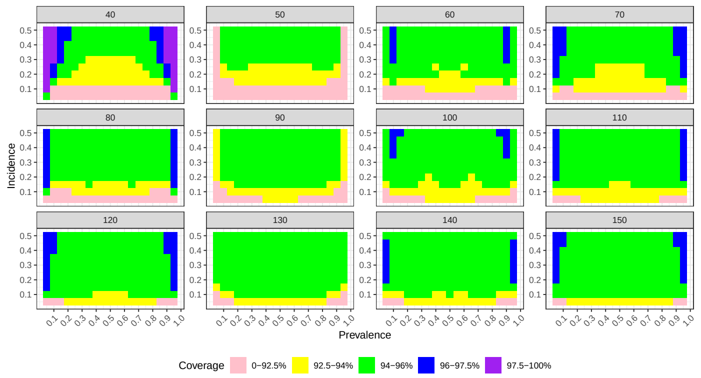
Source: Tipton, E., & Pustejovsky, J. E. (2015). Small-sample adjustments for tests of moderators and model fit using robust variance estimation in meta-regression. *Journal of Educational and Behavioral Statistics, 40*(6), 604-634.

12.2.3 Example: Heat maps of coverage

The visualization below shows the coverage of parametric bootstrap confidence intervals for momentary time sampling data In this simulation study the authors were comparing maximum likelihood estimators to posterior mode (penalized likelihood) estimators of prevalence. We have a 2-dimensional parameter space of prevalence (19 levels) by incidence (10 levels). We also have 15 levels of sample size.

One option here is to use a heat map, showing the combinations of prevelance and incidence as a grid for each sample size level. We break coverage into ranges of interest, with green being "good" (near 95%) and yellow being "close" (92.5% or above). For this to work, we need our MCSE to be small enough that our coverage is estimated precisely enough to show structure.





To see this plot IRL, see Pustejovsky, J. E., & Swan, D. M. (2015). Four methods for analyzing partial interval recording data, with application to single-case research. *Multivariate Behavioral Research, 50*(3), 365-380.

12.3 Modeling

Simulations are designed experiments, often with a full factorial structure. We can therefore leverage classic means for analyzing such full factorial experiment. In particular, we in effect model how a performance measure varies as a function of the different experimental factors. We can use regression or other modeling to do this.

First, in the language of a full factor experiment, we might be interested in the "main effects" or "interaction effects." A main effect is whether, averaging across the other factors in our experiment, a factor of interest systematically impacts our peformance measure. When we look at a main effect, the other factors help ensure our main effect is generalizable: if we see a trend when we average over the other varying aspects, then we can state that for a host of simulation contexts, grouped by levels of our main effect, we see a trend.

For example, consider the Bias of biserial correlation estimate from an extreme groups design example from above. Visually, we see that most factors appear to matter for bias, but we might want to get a sense of how much. In particular, does the the population vs sample cutoff option matter, on average, for bias?

```
options(scipen = 5)
mod = lm( bias ~ fixed + rho + I(rho^2) + p1 + n, data = r_F)
summary(mod, digits=2)
```

```
##
## Call:
## lm(formula = bias ~ fixed + rho + I(rho^2) + p1 + n, data = r_F)
##
## Residuals:
##        Min         1Q     Median         3Q
## -0.0215935 -0.0013608  0.0003823  0.0015677
##        Max
##  0.0081802
##
## Coefficients:
##                       Estimate  Std. Error
## (Intercept)         0.00218473  0.00015107
## fixedSample cutoff -0.00363520  0.00009733
## rho                -0.00942338  0.00069578
## I(rho^2)            0.00720857  0.00070868
## p1.L                0.00461700  0.00010882
## p1.Q               -0.00160546  0.00010882
## p1.C                0.00081464  0.00010882
## p1^4               -0.00011190  0.00010882
## n.L                 0.00362949  0.00010882
## n.Q                -0.00103981  0.00010882
## n.C                 0.00027941  0.00010882
## n^4                 0.00001976  0.00010882
##                    t value Pr(>|t|)
## (Intercept)         14.462  < 2e-16 ***
## fixedSample cutoff -37.347  < 2e-16 ***
## rho                -13.544  < 2e-16 ***
## I(rho^2)            10.172  < 2e-16 ***
## p1.L                42.426  < 2e-16 ***
## p1.Q               -14.753  < 2e-16 ***
## p1.C                 7.486 8.41e-14 ***
## p1^4                -1.028   0.3039
## n.L                 33.352  < 2e-16 ***
## n.Q                 -9.555  < 2e-16 ***
## n.C                  2.568   0.0103 *
## n^4                  0.182   0.8559
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.003372 on 4788 degrees of freedom
## Multiple R-squared:  0.5107, Adjusted R-squared:  0.5096
## F-statistic: 454.4 on 11 and 4788 DF,  p-value: < 2.2e-16
```

The above printout gives main effects for each factor, averaged across other factors. It is automatically generating linear, quadradic, cubic and fourth order contrasts for the ordered factors of p1 and n. We see that, across other contexts, the sample cutoff is around 0.004 lower than population.

We next discuss two additional tools:

- ANOVA can be useful for understanding major sources of variation in simulation results (e.g., identifying which factors have negligible/minor influence on the bias of an estimator).
- Smoothing (e.g., local linear regression) over continuous factors

```r
anova_table <- aov(bias ~ rho * p1 * fixed * n, data = r_F)
summary(anova_table)
```

```
##                   Df   Sum Sq  Mean Sq  F value
## rho                1 0.002444 0.002444  1673.25
## p1                 4 0.023588 0.005897  4036.41
## fixed              1 0.015858 0.015858 10854.52
## n                  4 0.013760 0.003440  2354.60
## rho:p1             4 0.001722 0.000431   294.71
## rho:fixed          1 0.003440 0.003440  2354.69
## p1:fixed           4 0.001683 0.000421   287.98
## rho:n              4 0.002000 0.000500   342.31
## p1:n              16 0.019810 0.001238   847.51
## fixed:n            4 0.013359 0.003340  2285.97
## rho:p1:fixed       4 0.000473 0.000118    80.87
## rho:p1:n          16 0.001470 0.000092    62.91
## rho:fixed:n        4 0.002929 0.000732   501.23
## p1:fixed:n        16 0.001429 0.000089    61.12
## rho:p1:fixed:n    16 0.000429 0.000027    18.36
## Residuals       4700 0.006866 0.000001
##                 Pr(>F)
## rho             <2e-16 ***
## p1              <2e-16 ***
## fixed           <2e-16 ***
## n               <2e-16 ***
## rho:p1          <2e-16 ***
## rho:fixed       <2e-16 ***
## p1:fixed        <2e-16 ***
## rho:n           <2e-16 ***
## p1:n           <2e-16 ***
## fixed:n        <2e-16 ***
## rho:p1:fixed   <2e-16 ***
## rho:p1:n       <2e-16 ***
## rho:fixed:n    <2e-16 ***
## p1:fixed:n     <2e-16 ***
## rho:p1:fixed:n <2e-16 ***
## Residuals
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
library(lsr)
etaSquared(anova_table)
```

```
##                     eta.sq eta.sq.part
## rho            0.021971037  0.26254289
## p1             0.212004203  0.77453319
## fixed          0.142527898  0.69783705
## n              0.123670355  0.66710072
## rho:p1         0.015479114  0.20052330
## rho:fixed      0.030918819  0.33377652
## p1:fixed       0.015125570  0.19684488
## rho:n          0.017979185  0.22560369
## p1:n           0.178055588  0.74260975
## fixed:n        0.120065971  0.66049991
## rho:p1:fixed   0.004247472  0.06439275
## rho:p1:n       0.013216569  0.17638308
## rho:fixed:n    0.026326074  0.29902214
## p1:fixed:n     0.012839790  0.17222072
## rho:p1:fixed:n 0.003857877  0.05883389
```

Perhaps we need an example where some things don't matter? We need to discuss what one learns from this table.—Miratrix

Chapter 13

Ensuring reproducibility

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

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

```
##          criterion         est        MCSE
## 1      alpha bias -0.02181369 0.025173513
## 2      alpha RMSE  0.07860782 0.005927365
## 3 V relative bias  0.68820771 0.142023637
## 4        coverage  0.80000000 0.068920244
```

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

```
##          criterion         est       MCSE
## 1      alpha bias -0.07737343 0.03992368
## 2      alpha RMSE  0.14258945 0.01028178
## 3 V relative bias  0.40659026 0.12055579
## 4        coverage  0.50000000 0.06892024
```

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

This is more possible than it sounds: Monte Carlo simulations are random, but computers are not. When we generate "random numbers" they actually come from a chain of mathematical equations that, given a number, will generate the next number in a deterministic sequence. Given that number, it will generate the next, and so on. The numbers we get back are a part of this chain of (very large) numbers that, ideally, cycles through an extremely long list of numbers in a haphazard and random looking fashion.

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

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

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

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

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

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

13.1 Seeds and pseudo-random number generators

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

Compare this:

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

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

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

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

To this:

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

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

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

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

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

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

13.2 Including seed in our simulation driver

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

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

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

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

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

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

13.3 Reasons for setting the seed

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

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

Chapter 14

Optimizing code (and why you often shouldn't)

Optimizing code is when you spend a bit more human effort to write code that will run faster on your computer. In some cases, this can be a critical boost to running a simulation, where you inherently will be doing things a lot of times. Cutting runtime down will always be tempting, as it allows you to run more replicates and get more precisely estimated performance measures for your simulation.

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

In the next sections we will look at a few optimization efforts applied to the ANOVA example in the prior chapters.

14.1 Hand-building functions

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

Following the formulas on p. 129 of Brown and Forsythe (1974) we have (using data as generated in Chapter **?**):

```
ANOVA_F <- function(sim_data) {

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

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

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

  return(pval)
}

ANOVA_F(sim_data)
```

```
## [1] 0.009208908
```

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

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

```
## Unit: microseconds
##       expr     min       lq     mean   median
##  Rfunction 518.507 525.6695 593.5698 531.4945
##     direct 232.335 240.5565 261.5123 245.9760
##        uq      max neval cld
##  573.4865 3522.609   100   b
##  263.9810  520.105   100  a
```

The direct function is 2.3 times faster than the built-in R function.

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

14.2 Computational efficiency versus simplicity

An alternative approach to having a function that, for each call, generates a single set of data would be to write a function that generates *multiple* sets of simulated data all at once.

For example, for our ANOVA example we could specify that we want `R` replications of the study and have the function spit out a matrix with `R` columns, one for each simulated dataset:

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

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

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

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

  return(sim_data)
}

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

```
## $group
##  [1] 1 1 1 2 2 2 2 2 2 3 3 4 4 4 4
##
## $x_mat
##             [,1]      [,2]       [,3]       [,4]
##  [1,]  2.1639323  1.646791 -0.9445957  1.2050393
##  [2,] -2.8225439  3.289125  0.8863700  3.7816431
##  [3,]  1.3419979 -3.020363  1.6196826  0.6425271
##  [4,] -0.5939495  1.184961  1.0200117  2.8301117
##  [5,]  0.1054279  3.499446  1.0267897  2.3166570
##  [6,]  4.0618127  4.340879  4.1291935 -1.0532910
##  [7,]  2.0611736  2.544518  1.3772973 -0.3861088
##  [8,]  2.2921786  2.299136  0.1706634  2.0329848
##  [9,]  2.5000308  3.686171  1.8712605  2.8557713
## [10,]  4.0966732  5.824634  6.6448117  2.6354699
## [11,]  5.9111997  3.480771  2.9453236  4.1680932
## [12,]  6.2201594  6.116809  5.8446326  6.2140816
## [13,]  5.9843999  6.488239  7.1954303  6.6280499
## [14,]  6.3928820  4.699057  5.9033074  8.8796297
## [15,]  7.9611464  7.282270  5.9095360  5.9131754
```

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

14.3 Reusing code to speed up computation

Again, computational and programming efficiency is usually a secondary consideration when you're starting to design a simulation study. It's better to produce accurate code, even if it's a bit slow, than to write code that is speedy but hard to follow (or even worse, that produces incorrect results). All that said, there is some glaring redundancy in the two functions used for the ANOVA simulation. Both `ANOVA_F` and `Welch_F` start by taking the simulated data and calculating summary statistics for each group, using the following code:

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

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

```r
summarize_data <- function(sim_data) {

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

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

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

```
## # A tibble: 4 x 4
##   group x_bar  s_sq     n
##   <int> <dbl> <dbl> <int>
## 1     1  1.90 1.42      3
## 2     2  1.82 2.10      6
## 3     3  4.40 0.626     2
## 4     4  6.58 0.485     4
```

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

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

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

  return(pval)
}

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

```
## [1] 0.0003129849
```

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

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

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

  return(pval)
}

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

```
## [1] 0.0003129849
```

The results are the same as before.

We should always test any optimized code against something we know is stable, since optimization is an easy way to get bad bugs. Here we check against R's implementation:

```
summary_stats <- summarize_data(sim_data)
F_results <- with(summary_stats,
                  ANOVA_F_agg(x_bar = x_bar, s_sq = s_sq, n = n))
aov_results <- oneway.test(x ~ factor(group), data = sim_data,
                           var.equal = TRUE)
all.equal(aov_results$p.value, F_results)
```

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

```
W_results <- with(summary_stats,
                  Welch_F_agg( x_bar = x_bar,
                               s_sq = s_sq, n = n))
aov_results <- oneway.test(x ~ factor(group),
                           data = sim_data,
                           var.equal = FALSE)
all.equal(aov_results$p.value, W_results)
```

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

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

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

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

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

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

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

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

```
one_run_fast <- function( mu, sigma_sq, sample_size ) {
  sim_data <- generate_data(mu = mu, sigma_sq = sigma_sq,
                            sample_size = sample_size)
  summary_stats <- summarize_data(sim_data)
  anova_p <- with(summary_stats,
                  ANOVA_F_agg(x_bar = x_bar,s_sq = s_sq, n = n))
  Welch_p <- with(summary_stats,
                  Welch_F_agg(x_bar = x_bar, s_sq = s_sq, n = n))
  tibble(ANOVA = anova_p, Welch = Welch_p)
}

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

```
## # A tibble: 1 x 2
##      ANOVA  Welch
##      <dbl>  <dbl>
## 1 0.000982 0.0209
```

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

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

```
## Unit: milliseconds
##   expr      min       lq     mean   median
##  noagg 1.317099 1.434244 1.792840 1.463407
##    agg 3.050497 3.284607 3.490857 3.353867
##        uq      max neval cld
##  1.531627 24.11427   100  a
##  3.539951 10.61070   100   b
```

And our relative speedup is:

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

```
## [1] 0.5
```

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

1. Code reuse is generally good because when you have the same code in multiple places it can make it harder to read and understand your code. If you see two blocks of code you might worry they are only mostly similar, not exactly similar, and waste time trying to differentiate. If you have a single, well-named function, you immediately know what a block of code is doing.
2. Saving the results of calculations can speed up your computation since you are saving your partial work. This can be useful to reduce calculations that are particularly time intensive.

Chapter 15

Error trapping and other headaches

If you have an advanced estimator, or are relying on some package, it is quite possible that every so often your estimate will trigger an error, or give you a NA result, or something similarly bad. More innocuous, you might have estimators that can generate warnings; if you run 10,000 trials, that can add up to a lot of warnings, which can be overwhelming to sort through. In some cases, these warnings might be coupled with the estimator returning a very off result; it is unclear, in this case, whether we should include that result in our overall performance measures for that estimator. After all, it tried to warn us!

In this section, we talk about some ways to make your simulations safe and robust, and also discuss some ways to track warnings and include them in performance measures.

15.1 Safe code

Sometimes if you write a function that does a lot of complex things with uncertain objects – i.e. consider a complex estimator using an unstable package not of your own creation – you can run into trouble where you have a intermittent error.

This can really be annoying; consider the case where your simulation crashes on 1 out of 500 chance: if you run 1000 simulation trials, your program will likely not make it to the end, thus wasting all of your time. To protect yourself, you can write code that can, instead of stopping when it reaches an error, trap the error and move on with the next simulation trial.

To illustrate, consider the following broken function that sometimes gives us what we want, sometimes gives us a NaN due to taking the square root of a negative number, and sometimes crashes completely due to `broken_code()` not existing:

```
my_complex_function = function( param ) {

    vals = rnorm( param, mean = 0.5 )
    if ( sum( vals ) > 5 ) {
        broken_code( 4 )
    } else {
        sqrt( sum( vals ) * sign( vals )[[1]] )
    }
}
```

We run it like so:

```
my_complex_function( 2 )
```

```
## [1] 0.9437257
```

```
my_complex_function( 2 )
```

```
## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced
```

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

```
my_complex_function( 7 )
```

```
## [1] 1.419031
```

So far so good, other than a NaN warning. Now let's run it a bunch of times:

```
resu = map( 1:20, ~ my_complex_function( 7 ) )
```

```
## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced
```

```
## Error in `map()`:
## i In index: 2.
## Caused by error in `broken_code()`:
## ! could not find function "broken_code"
```

Oh no! Our function crashes sometimes. To trap the errors we use the purrr package to make a "safe" function as so:

```
my_safe_function = safely( my_complex_function,
                           otherwise = NA )
my_safe_function( 7 )
```

```
## $result
## [1] NA
##
## $error
## <simpleError in broken_code(4): could not find function "broken_code">
```

Our safe version of the function gives us back a list of things: the result (or NULL if there was an error), and the error message (or NULL if there was no error). `safely()` is an example of a *function wrapper*, which takes a function and gives you back a new function that does something slightly different.

We include `otherwise = NA` so we always get a result, even if it is a NA result. Otherwise we would get a NULL when there is an error, which might be harder to track.

For example, we can use the above repeatedly, and then do a nice trick to get a list of error message separate from our list of results:

```
resu = map( 1:20, ~ my_safe_function( 7 ) )
```

```
## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced

## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced

## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced

## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced
```

```
resu <- transpose( resu )
unlist( resu$result )
```

```
##  [1]       NaN        NA       NaN 0.4354079
##  [5] 1.0734109        NA 1.5196734        NA
##  [9]       NaN 0.6382972 1.0836506        NA
## [13]        NA 1.6244187 1.1915743       NaN
## [17] 2.1165066 2.0638450 1.6465782        NA
```

The `transpose()` method takes a list of lists, and reorganizes them to give you a list of all the first elements, a list of all the second elements, etc. This is very powerful for wrangling data, because then we can make a tibble with list columns as so:

```
tb <- tibble( result = unlist( resu$result ),
        error = resu$error )
print( tb, n = 4 )
```

```
## # A tibble: 20 x 2
##    result error
##     <dbl> <list>
## 1 NaN     <NULL>
## 2  NA     <smplErrr>
## 3 NaN     <NULL>
## 4   0.435 <NULL>
## # i 16 more rows
```

There are other function wrappers in this "safe computing" family, such as "possibly," which will try to run something and give you a default value if it fails:

```
my_possible_function = possibly( my_complex_function,
                                 otherwise = NA )
my_possible_function( 7 )
```

```
## [1] 1.960578
```

```
rs <- map_dbl( 1:10, ~ my_possible_function(7) )
```

```
## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced

## Warning in sqrt(sum(vals) * sign(vals)[[1]]):
## NaNs produced
```

```
rs
```

```
##  [1] 1.197171       NA 1.533298      NaN       NA
##  [6]       NA       NA       NA      NaN 2.149232
```

There is also "quietly," which makes warnings and messages get bundled, rather than printed to the console:

```
my_quiet_function = quietly( my_complex_function )

my_quiet_function( 1 )
```

```
## $result
## [1] 0.3686244
##
## $output
## [1] ""
##
## $warnings
## character(0)
##
## $messages
## character(0)
```

This can be especially valuable to control massive amounts of printout in a simulation. If you have lots of extraneous printout, it can slow down the execution of your code far more than you might think.

Note that `quietly()` does not trap errors, just warnings:

```
rs <- map( 1:20, ~ my_quiet_function(7) )
```

```
## Error in `map()`:
## i In index: 3.
## Caused by error in `broken_code()`:
## ! could not find function "broken_code"
```

You can "double wrap" your function, if you want, but what you get back is a bit of a mess:

```
my_safe_quiet_function = quietly( safely( my_complex_function, otherwise = NA ) )
my_safe_quiet_function(7)
```

```
## $result
## $result$result
## [1] NA
##
## $result$error
## <simpleError in broken_code(4): could not find function "broken_code">
##
##
## $output
## [1] ""
##
## $warnings
## character(0)
##
## $messages
## character(0)
```

15.1.1 Making a safe function call

You can do some magical massaging to make things work better upfront so you can capture both errors and warnings and get the results in a nice tidy tibble. For example, say our `my_complex_function()` is a method designed to analyze our data. We might then write a wrapper that cleans up our safe and quiet version

```
unpack_mess <- function( mess ) {
  rs = tibble( result = NA, error = NA,
               warnings = "" )
  rs$result = mess$result$result
  rs$error = list( mess$result$error )
  if ( length( mess$warnings ) > 0 ) {
    rs$warnings = paste( mess$warnings, collapse="; " )
  }
  rs
}
unpack_mess( my_safe_quiet_function( 7 ) )
```

```
## # A tibble: 1 x 3
##   result error  warnings
##    <dbl> <list> <chr>
## 1   1.56 <NULL> ""
```

```
my_safe_quiet_function <- compose( unpack_mess, my_safe_quiet_function )
my_safe_quiet_function(7)
```

```
## # A tibble: 1 x 3
##   result error  warnings
##    <dbl> <list> <chr>
## 1   2.05 <NULL> ""
```

The `compose()` method makes a new function that will call `unpack_mess()` automatically. We have now wrapped our original function, in effect, three times. The result is a new function that takes the same parameters as our original `my_complex_function()` method, and returns some results, including information about errors and warnings, in a nicely organized way.

We could use such a modified function in a loop, and stack our results:

```
rs <- map_df( 1:20, ~ my_safe_quiet_function(6) )
rs
```

```
## # A tibble: 20 x 3
##    result error      warnings
##     <dbl> <list>     <chr>
##  1   1.76 <NULL>     ""
##  2   2.11 <NULL>     ""
##  3   1.68 <NULL>     ""
##  4 NaN    <NULL>     "NaNs produced"
##  5  NA    <smplErrr> ""
##  6   1.66 <NULL>     ""
##  7  NA    <smplErrr> ""
##  8   1.34 <NULL>     ""
##  9   1.35 <NULL>     ""
## 10   1.94 <NULL>     ""
## 11   2.15 <NULL>     ""
## 12   1.79 <NULL>     ""
## 13  NA    <smplErrr> ""
## 14 NaN    <NULL>     "NaNs produced"
## 15   1.91 <NULL>     ""
## 16   1.67 <NULL>     ""
## 17 NaN    <NULL>     "NaNs produced"
## 18   2.13 <NULL>     ""
## 19   2.10 <NULL>     ""
## 20 NaN    <NULL>     "NaNs produced"
```

These tools allow us to take existing analytic code and make it safe and quiet, so we can keep things organized in our simulation.

15.1.2 What to do with warnings in simulations

Sometimes our analytic strategy might give some sort of warning (or fail altogether). For example, from the cluster randomized experiment case study we have:

```r
set.seed(101012)  # (I picked this to show a warning.)
dat = gen_dat_model( J = 50, n_bar = 100, sigma2_u = 0 )
mod <- lmer( Yobs ~ 1 + Z + (1|sid), data=dat )
```

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

We have to make a deliberate decision as to what to do about this:

- Keep these “weird” trials?
- Drop them?

If you decide to drop them, you should drop the entire simulation iteration including the other estimators, even if they worked fine! If there is something particularly unusual about the dataset, then dropping for one estimator, and keeping for the others that maybe didn’t give a warning, but did struggle to estimate the estimand, would be unfair: in the final performance measures the estimators that did not give a warning could be being held to a higher standard, making the comparisons between estimators biased.

If your estimators generate warnings, you should calculate the rate of errors or warning messages as a performance measure. Especially if you drop some trials, it is important to see how often things are acting pecularly.

The main tool for doing this is the `quietly()` function:

```r
quiet_lmer = quietly( lmer )
qmod <- quiet_lmer( Yobs ~ 1 + Z + (1|sid), data=dat )
qmod
```

```
## $result
## Linear mixed model fit by REML ['lmerModLmerTest']
## Formula: ..1
##    Data: ..2
## REML criterion at convergence: 6485.293
## Random effects:
##  Groups   Name        Std.Dev.
##  sid      (Intercept) 0.0000  
##  Residual             0.9879  
## Number of obs: 2302, groups:  sid, 50
## Fixed Effects:
## (Intercept)            Z  
##    -0.03143      0.03433  
## optimizer (nloptwrap) convergence code: 0 (OK) ; 0 optimizer warnings; 1 lme4 warni
## 
## $output
## [1] ""
## 
## $warnings
## character(0)
## 
## $messages
## [1] "boundary (singular) fit: see help('isSingular')\n"
```

You then might have, in your analyzing code:

```
analyze_data <- function( dat ) {

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

    # Compile our results
    tibble( ATE_hat = coef(M1)["Z"],
            SE_hat = se.coef(M1)["Z"],
            message = message1,
            warning = warning1 )
}
```

Now you have your primary estimates, and also flags for whether there was a convergence issue. In the analysis section you can then evaluate what proportion of the time there was a warning or message, and then do subset analyses to those simulation trials where there was no such warning.

15.2 Protecting your functions with “stop”

When writing functions, especially those that take a lot of parameters, it is often wise to include `stopifnot()` statements at the top to verify the function is getting what it expects. These are sometimes called “assert statements” and are a tool for making errors show up as early as possible. For example, look at this (fake) example of generating data with different means and variances

```
make_groups <- function( means, sds ) {
  Y = rnorm( length(means), mean=means, sd = sds )
  round( Y )
}
```

If we call it, but provide different lengths for our means and variances, nothing happens, because R simply recycles the standard deviation parameter:

```
make_groups( c(100,200,300,400), c(1,100,10000) )
```

```
## [1]  100  133 1675  402
```

If this function was used in our data generating code, we would see the warnings but might not know exactly what they were. We can instead protect our function by putting an *assert statements* in our function like this:

```
make_groups <- function( means, sds ) {
  stopifnot( length(means) == length(sds) )
  Y = rnorm( length(means), mean=means, sd = sds )
  round( Y )
}
```

This ensures your code is getting called as you intended. What is nasty about this possible error is nothing is telling you something is wrong! You could build an entire simulation on this, not realizing that your fourth group has the variance of your first, and get results that make no sense to you. You could even publish something based on a finding that depends on this error, which would eventually be quite embarrasing.

These statements can also serve as a sort of documentation as to what you expect. Consider, for example:

```
make_xy <- function( N, mu_x, mu_y, rho ) {
  stopifnot( -1 <= rho && rho <= 1 )
  X = mu_x + rnorm( N )
  Y = mu_y + rho * X + sqrt(1-rho^2)*rnorm(N)
  tibble(X = X, Y=Y)
}
```

Here we see that rho should be between -1 and 1 quite clearly. A good reminder of what the parameter is for.

This also protects you from inadvetently misremembering the order of your parameters when you call the function (although it is good practice to name your parameters as you pass). Consider:

```r
a <- make_xy( 10, 2, 3, 0.75 )
b <- make_xy( 10, 0.75, 2, 3 )
```

```
## Error in make_xy(10, 0.75, 2, 3): -1 <= rho && rho <= 1 is not TRUE
```

```r
c <- make_xy( 10, rho = 0.75, mu_x = 2, mu_y = 3 )
```

Chapter 16

Saving files and results

Always save your simulation results to a file. Simulations are painful and time consuming to run, and you will invariably want to analyze the results of them in a variety of different ways, once you have looked at your preliminary analysis. We advocate saving your simulation as soon as it is complete. But there are some ways to do better than that, such as saving as you go. This can protect you if your simulation occasionally crashes, or if you want to rerun only parts of your simulation for some reason.

16.1 Saving simulations in general

Once your simulation has completed, you can save it like so:

```r
dir.create("results", showWarnings = FALSE )
write_csv( res, "results/simulation_CRT.csv" )
```

`write_csv()` is a tidyverse file-writing command; see "R for Data Science" textbook, 11.5.

You can then load it, just before analysis, as so:

```r
res = read_csv( "results/simulation_CRT.csv" )
```

There are two general tools for saving. The `read/write_csv` methods save your file in a way where you can open it with a spreadsheet program and look at it. But your results should be in a vanilla format (non-fancy data frame without list columns).

Alternatively, you can use the `saveRDS()` and `readRDS()` methods; these save objects to a file such that when you load them, they are as you left them. (The simpler format of a csv file means your factors, if you have them, may not preseve as factors, and so forth.)

16.2 Saving simulations as you go

If you are not sure you have time to run your entire simulation, or you think your computer might crash half way through, or something similar, you can save each chunk you run as you go, in its own file. You then stack those files at the end to get your final results. With clever design, you can even then selectively delete files to rerun only parts of your larger simulation—but be sure to rerun everything from scratch before you run off and publish your results, to avoid embarrassing errors.

Here, for example, is a script from a research project examining how one might use post-stratification to improve the precision of an IV estimate. This is the script that runs the simulation. Note the sourcing of other scripts that have all the relevant functions; these are not important here. Due to modular programming, we can see what this script does, even without those detail.

```r
source( "pack_simulation_functions.R" )

if ( !file.exists("results/frags" ) ) {
    dir.create("results/frags")
}

# Number of simulation replicates per scenario
R = 1000

# Do simulation breaking up R into this many chunks
M_CHUNK = 10

###### Set up the multifactor simulation #######

# chunkNo is a hack to make a bunch of smaller chunks for doing parallel more
# efficiently.
factors = expand_grid( chunkNo = 1:M_CHUNK,
                       N = c( 500, 1000, 2000 ),
                       pi_c = c( 0.05, 0.075, 0.10 ),
                       nt_shift = c( -1, 0, 1 ),
                       pred_comp = c( "yes", "no" ),
                       pred_Y = c( "yes", "no" ),
                       het_tx = c( "yes", "no" ),
                       sd0 = 1
                       )
factors <- factors %>% mutate(
    reps = R / M_CHUNK,
    seed = 16200320 + 1:n()
)
```

This generates a data frame of all our factor combinations. This is our list of "tasks" (each row of factors). These tasks have repeats: the "chunks" means we do a portion of each scenario, as specified by our simulation factors, as a process. This would allow for greater parallelization (e.g., if we had more cores), and also lets us save our work without finishing an entire scenario of, in this case, 1000 iterations.

To set up our simulation we make a little helper method to do one row. With each row, once we have run it, we save it to disk. This means if we kill our simulation half-way through, most of the work would be saved. Our function is then going to either do the simulation (and save the result to disk immediately), or, if it can find the file with the results from a previous run, load those results from disk:

```
safe_run_sim = safely( run_sim )
file_saving_sim = function( chunkNo, seed, ... ) {
    fname = paste0( "results/frags/fragment_", chunkNo, "_", seed, ".rds" )
    res = NA
    if ( !file.exists(fname) ) {
        res = safe_run_sim( chunkNo=chunkNo, seed=seed, ... )
        saveRDS(res, file = fname )
    } else {
        res = readRDS( file=fname )
    }
    return( res )
}
```

Note how we wrap our core `run_sim` method in `safely`; it was crashing very occasionally, and so to make the code more robust, we wrapped it so we could see any error messages.

We next run the simulation. We shuffle the rows of our task list so that which process gets what task is randomized. If some tasks are much longer (e.g., due to larger sample size) then this will get balanced out across our processes.

We have an `if-then` structure to easily switch between parallel and nonparallel code. This makes debugging easier: when running in parallel, stuff printed to the console does not show until the simulation is over. Plus it would be all mixed up since multiple processes are working simultaneously.

This overall structure allows the researcher to delete one of the "fragment" files from the disk, run the simulation code, and have it just do one tiny piece of the simulation. This means the researcher can insert a `browser()` command somewhere inside the code, and debug the code, in the natural context of how the simulation is being run.

```
# Shuffle the rows so we run in random order to load balance.
factors = sample_n(factors, nrow(factors) )

if ( TRUE ) {
    # Run in parallel
    parallel::detectCores()

    library(future)
    library(furrr)

    #plan(multiprocess) # choose an appropriate plan from future package
    #plan(multicore)
    plan(multisession, workers = parallel::detectCores() - 2 )

    factors$res <- future_pmap(factors, .f = file_saving_sim,
                              .options = furrr_options(seed = NULL),
                              .progress = TRUE )

} else {
  # Run not in parallel, used for debugging
  factors$res <- pmap(factors, .f = file_saving_sim )
}

tictoc::toc()
```

Our method cleverly loads files in, or generates them, for each chunk. The seed setting ensures reproducibility. Once we are done, we need to clean up our results:

```
sim_results <-
    factors %>%
    unnest(cols = res)

# Cut apart the results and error messages
sim_results$sr = rep( c("res","err"), nrow(sim_results)/2)
sim_results = pivot_wider( sim_results, names_from = sr, values_from = res )

saveRDS( sim_results, file="results/simulation_results.rds" )
```

16.3 Dynamically making directories

If you are generating a lot of files, then you should put them somewhere. But where? It is nice to dynamically generate a directory for your files on fly. One way to do this is to write a function that will make any needed directory, if it doesn't exist, and then put your file in that spot. For example, you might have your own version of `write_csv` as:

```
my_write_csv <- function( data, path, file ) {

  if ( !dir.exists( here::here( path ) ) ) {
    dir.create( here::here( path ), recursive=TRUE )
  }
  write_csv( data, paste0( path, file ) )
}
```

This will look for a path (starting from your R Project, by taking advantage of the `here` package), and put your data file in that spot. If the spot doesn't exist, it will make it for you.

16.4 Loading and combining files of simulation results

Once your simulation files are all generated, the following code will stack them all into a giant set of results, assuming all the files are themselves data frames stored in RDS objects. This function will try and stack all files found in a given directory; for it to work, you should ensure there are no other files stored there.

```
load.all.sims = function( filehead="results/" ) {

  files = list.files( filehead, full.names=TRUE)

  res = map_df( files, function( fname ) {
    cat( "Reading results from ", fname, "\n" )
    rs = readRDS( file = fname )
    rs$filename = fname
    rs
  })
  res
}
```

You would use as so:

```
results = load.all.sims( filehead="raw_results/" )
```

Chapter 17

Coding tidbits

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

17.1 Ways of repeating yourself

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

17.2 Default arguments for functions

To generate code both easy to use and configure, use default arguments. For example,

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

my_function()
```

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

```r
my_function( 5 )
```

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

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

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

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

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

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

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

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

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

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

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

```r
generate_blocked_data( n_k = c( 3, 2 ) )
```

```
##    B         Y0       Y1
## 1 B1 -0.7482593 4.486772
## 2 B1 -0.2617085 6.082758
## 3 B1  1.3035417 6.383409
## 4 B2  0.8599834 6.039337
## 5 B2 -1.2766371 3.663485
```

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

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

```r
args( generate_blocked_data )
```

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

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

17.3 Testing and debugging code in your scripts

If you have an extended script with a list of functions, you might have a lot of code that runs each function in turn, so you can easily remind yourself of what it does, or what the output looks like. One way to keep this code around, but not have it run all the time when you run your script, is to put the code inside a "FALSE block," that might look like so:

```r
if ( FALSE ) {
  res <- my_function( 10, 20, 30 )
  res
  # Some notes as to what I want to see.

  sd( res )
  # This should be around 20
}
```

You can then, when looking at the script, paste the code inside the block into the console when you want to run it. If you source the script, however, it will not run at all, and thus your code will source faster and not print out any extraneous output.

17.4 Keep multiple files of code

Simulations have two general phases: generate your results and analyze your results. The ending of the first phase should be to save the generated results. The beginning of the second phase should then be to load the results from a file and analyze them. These phases can be in a seperate '.R' files This allows for easily changing how one analyzes an experiment without re-running the entire thing.

17.5 The source command and keeping things organized

Once you have your multifactor simulation, if it is a particularly complex one, you will likely have three general collections of code:

- Code for generating data
- Code for analyzing data
- Code for running a single simulation scenario

If each of these pieces is large and complex, you might consider putting them in three different .R files. Then, in your primary simulation, you would source these files. E.g.,

```
source( "pack_data_generators.R" )
source( "pack_estimators.R" )
source( "pack_simulation_support.R" )
```

You might also have `pack_simulation_support.R` source the other two files, and then source the single simulation support file in your primay file.

One reason for putting code in individual files is you can then have testing code in each of your files (in False blocks, like described above), testing each of your components. Then, when you are not focused on that component, you don't have to look at that testing code.

Another good reason for this type of modular organizing is you can then have a variety of data generators, forming a library of options. You can then easily create different simulations that use different pieces, in a larger project.

For example, in one recent simulation project on estimators for an Instrumental Variable analysis, we had several different data generators for generating different types of compliance patterns (IVs are often used to handle noncompliance in randomized experiments). Our data generation code file then had several methods:

```
> ls()
[1] "describe_sim_data"  "make_dat"           "make.dat.1side"
[4] "make.dat.1side.old" "make.dat.orig"      "make.dat.simple"
[7] "make.dat.tuned"     "rand.exp"           "summarize_sim_data"
```

The describe and summarize methods printed various statistics about a sample dataset; these are used to debug and understand how the generated data looks. We also had a variety of different DGP methods because we had different versions that came up as we were trying to chase down errors in our estimators and understand strange behavior.

Putting the estimators in a different file also had a nice additional purpose: we also had an applied data example in our work, and we could simply source that file and use those estimators on our actual data. This ensured our simulation and applied analysis were perfectly aligned in terms of the estimators we were using. Also, as we debugged our estimators and tweaked them, we immediately could re-run our applied analysis to update those results with minimal effort.

Modular programming is key.

17.6 Debugging with browser

Consider the code taken from a simulation:

```
if ( any( is.na( rs$estimate ) ) ) {
    browser()
}
```

The `browser()` command stops your code and puts you in an interactive console where you can look at different objects and see what is happening. Having it triggered when something bad happens (in this case when a set of estimates has an unexpected NA) can help untangle what is driving a rare event.

Chapter 18

Parallel Processing

Especially if you take our advice of "when in doubt, go more general" and if you calculate monte carlo standard errors, you will quickly come up against the limits of your computer. Simultions can be incredibly computationally intensive, and there are a few means for dealing with that. The first, touched on at times throughout the book, is to optimize ones code by looking for ways to remove extraneous calculation (e.g., by writing ones own methods rather than using the safety-checking and thus sometimes slower methods in R, or by saving calculations that are shared across different estimation approaches). The second is to use more computing power. This latter approach is the topic of this chapter.

There are two general ways to do parallel calculation. The first is to take advantage of the fact that most modern computers have multiple cores (i.e., computers) built in. With this approach, we tell R to use more of the processing power of your desktop or laptop. If your computer has eight cores, you can easily get a near eight-fold increase in the speed of your simulation.

The second is to use cloud computing, or compute on a cluster. A computing cluster is a network of hundreds or thousands of computers, coupled with commands where you break apart a simulation into pieces and send the pieces to your army of computers. Conceptually, this is the same as when you do baby parallel on your desktop: more cores equals more simulations per minute and thus faster simulation overall. But the interface to a cluster can be a bit tricky, and very cluster-dependent.

But once you get it up and running, it can be a very powerful tool. First, it takes the computing off your computer entirely, making it easier to set up a job to run for days or weeks without making your day to day life any more difficult. Second, it gives you hundreds of cores, potentially, which means a speed-up of hundreds rather than four or eight.

Simulations are a very natural choice for parallel computation. With a multifactor experiment it is very easy to break apart the overall into pieces. For example, you might send each factor combination to a single machine. Even without multi factor experiments, due to the cycle of "generate data, then analyze," it is easy to have a bunch of computers doing the same thing, with a final collection step where all the individual iterations are combined into one at the end.

18.1 Parallel on your computer

Most modern computers have multiple cores, so you can run a parallel simulation right in the privacy of your own home!

To assess how many cores you have on your computer, you can use the `detectCores()` method in the `parallel` package:

```
parallel::detectCores()
```

```
## [1] 8
```

Normally, unless you tell it to do otherwise, ***R only uses one core***. This is obviously a bit lazy on R's part. But it is easy to take advantage of multiple cores using the `future` and `furrr` packages.

```
library(future)
library(furrr)
```

In particular, the `furrr` package replicates our `map` functions, but in parallel. We first tell our R session what kind of parallel processing we want using the `future` package. In general, using `plan(multisession)` is the cleanest: it will start one entire R session per core, and have each session do work for you. The alternative, `multicore` does not seem to work well with Windows machines, nor with RStudio in general.

The call is simple:

```
plan(multisession, workers = parallel::detectCores() - 1 )
```

The `workers` parameter specifies how many of your cores you want to use. Using all but one will let your computer still operate mostly normally for checking email and so forth. You are carving out a bit of space for your own adventures.

Once you set up your plan, you use `future_pmap()`; it works just like `pmap()` but evaluates across all available workers specified in the plan call. Here we are running a parallel version of the multifactor experiment discussed in Chapter @ref(exp_design) (see chapter @ref(case_Cronback) for the simulation itself).

```
tictoc::tic()
params$res = future_pmap(params,
                         .f = run_alpha_sim,
                         .options = furrr_options(seed = NULL))
tictoc::tic()
```

Note the `.options = furrr_options(seed = NULL)` part of the argument. This is to silence some warnings. Given how tasks are handed out, R will get upset if you don't do some handholding regarding how it should set seeds for pseudoranom number generation. In particular, if you don't set the seed, the multiple sessions could end up having the same starting seed and thus run the exact same simulations (in principle). We have seen before how to set specific seed for each simulation scenario, but `furrr` doesn't know we have done this. This is why the extra argument about seeds: it is being explicit that we are handling seed setting on our own.

We can compare the running time to running in serial (i.e. using only one worker):

```
tictoc::tic()
params$res2 = dplyr::select(params, n:seed) %>%
  pmap(.f = run_alpha_sim)
tictoc::tic()
```

(The `select` command is to drop the `res` column from the parallel run; it would otherwise be passed as as parameter to `run_alpha_sim` which would in turn cause an error due to the unrecognized parameter.)

18.2 Parallel off your computer

In general, a "cluster" is a system of computers that are connected up to form a large distributed network that many different people can use to do large computational tasks (e.g., simulations!). These clusters will have some overlaying coordinating programs that you, the user, will interact with to set up a "job," or set of jobs, which is a set of tasks you want some number of the computers on the cluster to do for you in tandum.

These coordinating programs will differ, depending on what cluster you are using, but have some similarities that bear mention. For running simulations, you only need the smallest amount of knowledge about how to engage with these systems because you don't need all the individual computers working on your project communicating with each other (which is the hard part of distributed computing, in general).

18.2.1 What is a command-line interface?

In the good ol' days, when things were simpler, yet more difficult, you would interact with your computer via a "command-line interface." The easiest way to think about this is as an R console, but in a different language that the entire computer speaks. A command line interface is designed to do things like find files with a specific name, or copy entire directories, or, importantly, start different programs. Another place you may have used a command line inteface is when working with Git: anything fancy with Git is often done via command-line. People will talk about a "shell" (a generic term for this computer interface) or "bash" or "csh." You can get access to a shell from within RStudio by clicking on the "Terminal" tab. Try it, if you've never done anything like this before, and type

```
ls
```

It should list some file names. Note this command does *not* have the parenthesis after the command, like in R or most other programming languages. The syntax of a shell is usually mystifying and brutal: it is best to just steal scripts from the internet and try not to think about it too much, unless you want to think about it a lot.

Importantly for us, from the command line interface you can start an R program, telling it to start up and run a script for you. This way of running R is noninteractive: you say "go do this thing," and R starts up, goes and does it, and then quits. Any output R generates on the way will be saved in a file, and any files you save along the way will also be at your disposal once R has completed.

To see this in action make the following script in a file called "dumb_job.R":

```
library( tidyverse )
cat( "Making numbers\n" )
Sys.sleep(30)
cat( "Now I'm ready\n" )
dat = tibble( A = rnorm( 1000 ), B = runif( 1000 ) * A )
write_csv( dat, file="sim_results.csv" )
Sys.sleep(30)
cat( "Finished\n" )
```

Then open the terminal and type (the ">" is not part of what you type):

```
> ls
```

Do you see your `dumb_job.R` file? If not, your terminal session is in the wrong directory. In your computer system, files are stored in a directory structure, and when you open a terminal, you are somewhere in that structure.

To find out where, you can type

```
> pwd
```

for "Print Working Directory". Save your dumb job file to wherever the above says. You can also change directories using `cd`, e.g., `cd ~/Desktop/temp` means "change directory to the temp folder inside Desktop inside my home directory" (the `~` is shorthand for home directory). One more useful commands is `cd ..` (go up to the parent directory).

Once you are in the directory with your file, type:

```
> R CMD BATCH dumb_job.R R_output.txt --no-save
```

The above command says "Run R" (the first part) in batch mode (the "CMD BATCH" part), meaning source the `dumb_job.R` script as soon as R starts, saving all console output in the file `R_output.txt` (it will be saved in the current directory where you run the program), and where you don't save the workspace when finished.

This command should take about a minute to complete, because our script sleeps a lot (the sleep represents your script doing a lot of work, like a real simulation would do). Once the command completes (you will see your ">" prompt come back), verify that you have the `R_output.txt` and the data file `sim_results.csv` by typing `ls`. If you open up your Finder or Microsoft equivilent, you can actually see the `R_output.txt` file appear half-way through, while your job is running. If you open it, you will see the usual header of R telling you what it loading, the "Making numbers" comment, and so forth. R is saving everything as it works through your script.

Running R in this fashion is the key element to a basic way of setting up a massive job on the cluster: you will have a bunch of R programs all "going and doing something" on different computers in the cluster. They will all save their results to files (they will have files of different names, or you will not be happy with the end result) and then you will gather these files together to get your final set of results.

Small Exercise: Try putting an error in your `dumb_job.R` script. What happens when you run it in batch mode?

18.2.2 Running a job on a cluster

In the above, you can run a command on the command-line, and the command line interface will pause while it runs. As you saw, when you hit return with the above R command, the program just sat there for a minute before you got your command-line prompt back, due to the sleep.

When you properlly run a big job (program) on a cluster, it doesn't quite work that way. You will instead set a program to run, but tell the cluster to run it somewhere else (people might say "run in the background"). This is good because you get your command-line prompt back, and can do other things, while the program runs in the background.

There are various methods for doing this, but they usually boil down to a request from you to some sort of managerial process that takes requests and assigns some computer, somewhere, to do them. (Imagine a dispatcher at a taxi company.

You call up, ask for a ride, and it sends you a taxi to do it. The dispatcher is just fielding requests, assinging them to taxis.)

For example, one dispatcher is the slurm (which may or may not be on the cluster you are attempting to use; this is where a lot of this information gets very cluster-specific).

You first set up a script that describes the job to be run. It is like a work request. This would be a plain text file, such as this example (`sbatch_runScript.txt`):

```
#!/bin/bash
#SBATCH -n 32                                                   # Number of cores reque
#SBATCH -N 1                                                      # Ensure that all co
#SBATCH -t 480                                                  # Runtime in minutes
#SBATCH -p stats                                                # Partition to submit t
#SBATCH --mem-per-cpu=1000                    # Memory per cpu in MB
#SBATCH --open-mode=append                      # Append to output file, don't truncat
#SBATCH -o /output/directory/out/%j.out # Standard out goes to this file
#SBATCH -e /output/directory/out/%j.err # Standard err goes to this file
#SBATCH --mail-type=ALL                         # Type of email notification- BEGIN,EN
#SBATCH --mail-user=email@gmail.com         # Email address

# You might have some special loading of modules in the computing environment
source new-modules.sh
module load gcc/7.1.0-fasrc01
module load R
export R_LIBS_USER=$HOME/apps/R:$R_LIBS_USER

#R file to run, and txt files to produce for output and errors
R CMD BATCH estimator_performance_simulation.R logs/R_output_${INDEX_VAR}.txt --no-save
```

This file starts with a bunch of variables that describe how sbatch should handle the request. It then has a series of commands that get the computer environment ready. Finally, it has the `R CMD BATCH` command that does the work you want.

These scripts can be quite confusing to understand. There are so many options! What do these things even do? The answer is, for researchers early on their journey to do this kind of work, "Who knows?" The general rule is to find an example file for the system you are working on that works, and then modify it for your own purposes.

Once you have such a file, you could run it on the command line, like this:

```
sbatch -o stdout.txt \
        --job-name=my_script \
        sbatch_runScript.txt
```

You do this, and it will *not* sit there and wait for the job to be done. The `sbatch` command will instead send the job off to some computer which will do the work in parallel.

Interestingly, your R script could, at this point, do the "one computer" parallel type code listed above. Note the script above has 32 cores; your single job could then have 32 cores all working away on their individual pieces of the simulation, as before (e.g., with `future_pmap`). You would have a 32-fold speedup, in this case.

This is the core element to having your simulation run on a cluster. The next step is to do this *a lot*, sending off a bunch of these jobs to different computers.

Some final tips

- Remember to save a workspace or RDS!! Once you tell Odyssey to run an R file, it, well, runs the R file. But, you probably want information after it's done - like an R object or even an R workspace. For any R file you want to run on Odyssey, remember at the end of the R file to put a command to save something after everything else is done. If you want to save a bunch of R objects, an R workspace might be a good way to go, but those files can be huge. A lot of times I find myself wanting only one or two R objects, and RDS files are a lot smaller.
- Moving files from a cluster to your computer. You will need to first upload your files and code to the cluster, and then, once you've saved your workspace/RDS, you need those back on your computer. Using a scp client such as FileZilla is an easy way to do this file-transfer stuff. You can also use a Git repo for the code, but checking in the simulation results is not generally advisable: they are big, and not really in the spirit of a verson control system. Download your simulation results outside of Git, and keep your code in Git, is a good rule of thumb.

18.2.3 Checking on a job

Once your job is working on the cluster, it will keep at it until it finishes (or crashes, or is terminated for taking up too much memory or time). As it chugs away, there will be different ways to check on it. For example, you can, from the console, list the jobs you have running to see what is happening:

```
sacct -u lmiratrix
```

except, of course, "`lmiratrix`" would be changed to whatever your username is. This will list if your file is running, pending, timed out, etc. If it's pending, that usually means that someone else is hogging up space on the cluster and your job request is in a queue waiting to be assigned.

The `sacct` command is customizable, e.g.,

```
sacct -u lmiratrix --format=JobID,JobName%30,State
```

will not truncate your job names, so you can find them more easily.

You can check on a specific job, if you know the ID:

```
squeue -j JOBID
```

Something that's fun is you can check who's running files on the stats server by typing:

```
showq-slurm -p stats -o
```

You can also look at the log files

```
tail my_log_file.log
```

to see if it is logging information as it is working.

The email arguments, above, cause the system to email you before and after the job is complete. The email notifications you can choose are `BEGIN`, `END`, `FAIL`, and `ALL`; `ALL` is generally good. What is a few more emails?

18.2.4 Running lots of jobs on a cluster

We have seen how to fire off a job (possibly a big job) that can run over days or weeks to give you your results. There is one more piece that can allow you to use even more computing resources to do things even faster, which is to do a whole bunch of job requests like the above, all at once. This multiple dispatching of sbatch commands is the final component for large simulations on a cluster: you are setting in motion a bunch of processes, each set to a specific task.

Asking for multiple, smaller, jobs is also nicer for the cluster than having one giant job that goes on for a long time. By dividing a job into smaller pieces, and asking the scheduler to schedule those pieces, you can let the scheduler share and allocate resources between you and others more fairly. It can make a list of your jobs, and farm them out as it has space. This might go faster for you; with a really big job, the scheduler can't even allocate it until the needed number of workers is available. With smaller jobs, you can take a lot of little spaces to get your work done. Especially since simulation is so independent (just doing the same thing over and over) there is rarely any need for one giant process that has to do everything.

To make multiple, related, requests, we create a for-loop in the Terminal to make a whole series sbatch requests. Then, each sbatch request will do one part of the overall simulation. We can write this program in the shell, just like you can write R scripts in R. A shell scripts does a bunch of shell commands for you, and can even have variables and loops and all of that fun stuff.

For example, the following `run_full_simulation.sh` is a script that fires off a bunch of jobs for a simulation. Note that it makes a variable `INDEX_VAR`, and sets up a loop so it can run 500 tasks indexed 1 through 500.

The first `export` line adds a collection of R libraries to the path stored in `R_LIBS_USER` (a "path" is a list of places where R will look for libraries). The next line sets up a for loop: it will run the indented code once for each number from 1 to 500. The script also specifies where to put log files and names each job with the index so you can know who is generating what file.

```
export R_LIBS_USER=$HOME/apps/R:$R_LIBS_USER

for INDEX_VAR in $(seq 1 500); do

  #print out indexes
  echo "${INDEX_VAR}"

  #give indexes to R so it can find them.
  export INDEX_VAR

  #Run R script, and produce output files
  sbatch -o logs/sbout_p${INDEX_VAR}.stdout.txt \
        --job-name=runScr_p${INDEX_VAR} \
        sbatch_runScript.txt

  sleep 1 # pause to be kind to the scheduler

done
```

One question is then how do the different processes know what part of the simulation they should be working on? E.g., each worker needs to have its own seed so it don't do exactly the same simulation as a different worker! The workers also need their own filenames so they save things in their own files. The key is the `export INDEX_VAR` line: this puts a variable in the environment that will be set to a specific number. Inside your R script, you can get that index like so:

```
index <- as.numeric(as.character(Sys.getenv("INDEX_VAR")))
```

You can then use the index to make unique filenames when you save your results, so each process has its own filename:

```
filename = paste0( "raw_results/simulation_results_", index, _".rds" )
```

You can also modify your seed such as with:

```
factors = mutate( factors,
                  seed = set.seed( 1000 * seed + index ) )
```

Now even if you have a series of seeds within the simulation script (as we have seen before), each script will have unique seeds not shared by any other script (assuming you have fewer than 1000 separate job requests).

This still doesn't exactly answer how to have each worker know what to work on. Conider the case of our multifactor experiment, where we have a large combination of simulation trials we want to run.

There are two approaches one might use here. One simple approach is the following: we first generate all the factors with `expand_grid()` as usual, and then we take the row of this grid that corresponds to our index.

```
sim_factors = expand_grid( ... )
index <- as.numeric(as.character(Sys.getenv("INDEX_VAR")))
filename = paste0( "raw_results/simulation_results_", index, _".rds" )

stopifnot( index >= 1 && index <= nrow(sim_factors ) )
do.call( my_sim_function, sim_factors[ index, ] )
```

The `do.call()` command runs the simulation function, passing all the arguments listed in the targeted row. You then need to make sure you have your shell call the right number of workers to run your entire simulation.

One problem with this approach is some simulations might be a lot more work than others: consider your simulation with a huge sample size vs. one with a small sample size. Instead, you can have each worker run a small number of simulations of each scenario, and then stack your results later. E.g.,

```
sim_factors = expand_grid( ... )
index <- as.numeric(as.character(Sys.getenv("INDEX_VAR")))
sim_factors$seed = 1000000 * index + 17 * 1:nrow(sim_factors)
```

and then do your usual `pmap` call with `R = 10` (or some other small number of replicates.)

For saving files and then loading and combining them for analysis, see Section **?**.

18.2.5 Resources for Harvard's Odyssey

The above guidiance is tailored for Harvard' computing environment, primarily. For that environment in particular, there are many additional resources such as:

- Odyssey Guide: https://rc.fas.harvard.edu/resources/odyssey-quickstart-guide/
- R on Odyssey: https://rc.fas.harvard.edu/resources/documentation/software/r/

For installing R packages so they are seen by the scripts run by sbatch, see (https://www.rc.fas.harvard.edu/resources/documentation/software-on-odyssey/r/)

Other clusters should have similar documents giving needed guidance for their specific contexts.

18.2.6 Acknowledgements

Some of the above material is based on tutorials built by Kristen Hunter and Zach Branson, past doctoral students of Harvard's statistics department.

Chapter 19

Using simulation as a power calculator

We can use simulation as a power calculator. In particular, to estimate power, we generate data according to our best guess as to what we might find in a planned evaluation, and then analyze these synthetic data and see if we detect the effect we built into our DGP. We then do this repeatedly, and see how often we detect our effect. This is power.

Now, if we are generally right about our guesses about our DGP and the associated parameters we plugged into it, in terms of some planned study, then our power will be right on. This is all a power analysis is, using simulation or otherwise.

Simulation has benefits over using power calculators because we can take into account odd aspects of our modeling, and also do non-standard approaches to evaluation that we might not find in a normal power calculator.

We illustrate this idea with a case study. In this example, we are planning a school-level intervention to reduce rates of discipline via a socio-emotional targeting intervention on both teachers and students, where we have strongly predictive historic data and a time-series component. This is a planned RCT, where we will treat entire schools (so a cluster-randomized study). We are struggling because treating each school is very expensive (we have to run a large training and coaching of the staff), so each unit is a major decision. We want something like 4, 5, or maybe 6 treated schools. Our diving question is: Can we get away with this?

19.1 Getting design parameters from pilot data

We had pilot data from school administrative records (in particular discipline rates for each school and year for a series of five years), and we use those to estimate parameters to plug into our simulation. We assume our experimental sample will be on schools that have chronic issues with discipline, so we filtered our historic data to get schools we imagined to likely be in our study.

We ended up with the following data, with log-transformed discipline rates for each year (we did this to put things on a multiplicative scale, and to make our data more normal given heavy skew in the original). Each row is a potential school in the district.

```
datW = read_csv( "data/discipline_data.csv" )
```

```
## Rows: 27 Columns: 6
## -- Column specification --------------------------
## Delimiter: ","
## chr (1): Code
## dbl (5): 2015, 2016, 2017, 2018, 2019
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
datW
```

```
## # A tibble: 27 x 6
##    Code  `2015` `2016` `2017` `2018` `2019`
##    <chr>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 S1     -2.87  -2.81  -2.93  -3.52  -4.90
##  2 S2     -3.60  -2.83  -2.56  -2.76  -3.32
##  3 S3     -3.00  -2.88  -2.81  -3.39  -4.91
##  4 S4     -3.90  -3.20  -2.53  -3.67  -4.34
##  5 S5     -2.46  -2.00  -3.34  -3.66  -4.71
##  6 S6     -2.86  -2.74  -2.51  -3.21  -3.80
##  7 S7     -2.47  -2.59  -2.69  -2.15  -2.43
##  8 S8     -2.13  -1.93  -1.82  -2.21  -2.95
##  9 S9     -3.36  -3.16  -3.06  -3.26  -3.10
## 10 S10    -2.89  -2.54  -2.26  -2.89  -3.25
## # i 17 more rows
```

We use these to calculate a mean and covariance structure for generating data:

```
lpd_mns = apply( datW[,-1], 2, mean )
lpd_mns
```

```
##      2015      2016      2017      2018      2019
## -3.076298 -2.868337 -2.931562 -3.337221 -4.011440
```

```
lpd_cov = cov( datW[,-1] )
lpd_cov
```

```
##           2015      2016      2017      2018
## 2015 0.4191843 0.2996073 0.2282627 0.3691894
## 2016 0.2996073 0.3656335 0.2014201 0.2511376
## 2017 0.2282627 0.2014201 0.3084799 0.2927782
## 2018 0.3691894 0.2511376 0.2927782 0.5767486
## 2019 0.1921622 0.1623542 0.2541191 0.2927812
##           2019
## 2015 0.1921622
## 2016 0.1623542
## 2017 0.2541191
## 2018 0.2927812
## 2019 0.5425783
```

19.2 The data generating process

We then write a data generator that, given a desired number of control and treatment schools, and a treatment effect, makes a dataset by sampling vectors of discipline rates, and then imposes a "treatment effect" of scaling the discipline rate by the treatment coefficient for the last two years.

```
make_dat_param = function( n_c, n_t, tx=1 ) {
    n = n_c + n_t
    lpdisc = MASS::mvrnorm( n, mu = lpd_mns, Sigma = lpd_cov )
    lpdisc = exp( lpdisc )
    colnames( lpdisc ) = paste0( "pdisc_", colnames( lpdisc ) )
    lpdisc = as.data.frame( lpdisc ) %>%
        mutate( ID = 1:n(),
                Z = 0 + ( sample( n ) <= n_t ) )

    # Add in treatment effect
    lpdisc = mutate( lpdisc,
                     pdisc_2018 = pdisc_2018 * ifelse( Z == 1, tx, 1 ),
                     pdisc_2019 = pdisc_2019 * ifelse( Z == 1, tx, 1 ) )

    lpdisc %>%
      relocate( ID, Z )
}
```

Our function generates schools with discipline given by the provided mean and covariance structure; we have calibrated our data generating process to give us data that looks very similar to the data we would see in the field.

For our impact model, the treatment kicks in for the final two years, multiplying discipline rate by `tx` (so `tx = 1` means no treatment effect).

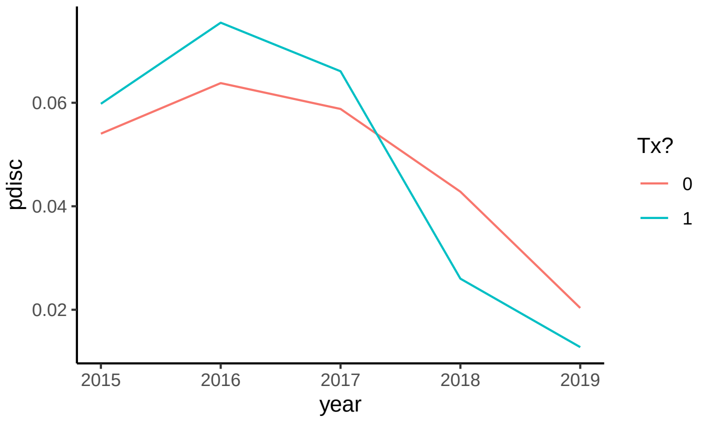
Testing our function gives this:

```
set.seed( 59585 )
a = make_dat_param( 100, 100, 0.5 )
head( a, n = 4 )
```

```
##   ID Z pdisc_2015 pdisc_2016 pdisc_2017
## 1  1 1 0.06881051 0.10758281 0.06382513
## 2  2 1 0.10945911 0.03779567 0.05328905
## 3  3 1 0.01649212 0.03527745 0.01868862
## 4  4 0 0.14944083 0.12577203 0.09451428
##   pdisc_2018  pdisc_2019
## 1 0.02737314 0.012127238
## 2 0.03669081 0.022538540
## 3 0.00256124 0.002767943
## 4 0.04844326 0.022815571
```

We can group each treatment arm and look at discipline over the years:

```
  aL = a %>%
        pivot_longer( pdisc_2015:pdisc_2019,
                      names_to = c( ".value", "year" ),
                      names_pattern = "(.*)_(.*)" ) %>%
        mutate( year = as.numeric( year ) )

    aLg = aL %>% group_by( year, Z ) %>%
        summarise( pdisc = mean( pdisc ) )
    ggplot( aLg, aes( year, pdisc, col=as.factor(Z) ) ) +
        geom_line() +
      labs( color = "Tx?" )
```



Our treatment group drops faster than the control. We see the nonlinear structure actually observed in our original data in terms of discipline over time has been replicated.

We next write some functions to analyze our data. This should feel very familiar: we are just doing our simulation framework, as usual.

```
eval_dat = function( sdat ) {

    # No covariate adjustment, average change model (on log outcome)
    M_raw = lm( log( pdisc_2018 ) ~ 1 + Z, data=sdat )

    # Simple average change model using 2018 as outcome.
    M_simple = lm( pdisc_2018 ~ 1 + Z + pdisc_2017 + pdisc_2016 + pdisc_2015,
                   data=sdat )

    # Simple model on logged outcome
    M_log = lm( log( pdisc_2018 ) ~ 1 + Z + log( pdisc_2017) + log( pdisc_2016) + log( pdisc_2015
                data=sdat )

    # Ratio of average disc to average prior disc as outcome
    sdat = mutate( sdat,
                   avg_disc = (pdisc_2018 + pdisc_2019)/2,
                   prior_disc = (pdisc_2017 + pdisc_2016 + pdisc_2015 )/3,
                   disc = pdisc_2018 / prior_disc,
                   disc_two = avg_disc / prior_disc )
    M_ratio = lm( disc ~ 1 + Z, data = sdat )
    M_ratio_twopost = lm( disc_two ~ 1 + Z, data = sdat )

    # Use average of two post-tx time periods, averaged to reduce noise
    M_twopost = lm( log( avg_disc ) ~ 1 + Z + log( pdisc_2017 ) + log( pdisc_2016 ) + log( pdisc_

    # Time and unit fixed effects
    sdatL = pivot_longer( sdat, cols = pdisc_2015:pdisc_2019,
                          names_to = "year",
                          values_to = "pdisc" ) %>%
        mutate( Z = Z * (year %in% c( "pdisc_2018", "pdisc_2019" ) ),
                ID = paste0( "S", ID ) )

    M_2wfe = lm( log( pdisc ) ~ 0 + ID + year + Z,
                 data=sdatL )

    # Bundle all our models by getting the estimated treatment impact
    # from each.
    models <- list( raw=M_raw, simple=M_simple,
                    log=M_log, ratio = M_ratio,
                    ratio_twopost = M_ratio_twopost,
                       log_twopost = M_twopost,
                       FE = M_2wfe )
    rs <- map_df( models, broom::tidy, .id="model" ) %>%
        filter( term=="Z" ) %>%
        dplyr::select( -term ) %>%
      arrange( model )

    rs
}
```

Our method marches through a host of models; we weren't sure what the gains would be from one model to another, so we decided to conduct power analyses on all of them. Again, we look at what our evaluation function does:

```
dat = make_dat_param( n_c = 4, n_t = 4, tx = 0.5 )
eval_dat( dat )
```

```
## # A tibble: 7 x 5
##   model        estimate std.error statistic p.value
##   <chr>           <dbl>     <dbl>     <dbl>   <dbl>
## 1 FE           -0.525     0.326     -1.61     0.119
## 2 log           0.208     0.229      0.909    0.430
## 3 log_twopost -0.0673    0.291     -0.232    0.832
## 4 ratio        -0.00850   0.119     -0.0714   0.945
## 5 ratio_twop~ -0.203     0.141     -1.44     0.201
## 6 raw           0.106     0.163      0.652    0.539
## 7 simple        0.00730   0.00625    1.17     0.327
```

We have a nice set of estimates, one for each model.

19.3 Running the simulation

Now we put it all together in our classic simulator:

```
sim_run = function( n_c, n_t, tx, R, seed = NULL ) {
    if ( !is.null( seed ) ) {
        set.seed(seed)
    }
    cat( "Running n_c, n_t =", n_c, n_t, "tx =", tx, "\n" )
    rps = rerun( R, {
        sdat = make_dat_param(n_c = n_c, n_t = n_t, tx = tx)
        eval_dat( sdat )
    })
    bind_rows( rps )
}
```

We then do the usual to run across a set of scenarios, running `sim_run` on each row of the following:

```
res = expand_grid( tx = c( 1, 0.75, 0.5 ),
                   n_c = c( 4, 5, 6, 8, 12, 20 ),
                   n_t = c( 4, 5, 6 ) )
res$R = 1000
res$seed = 1010203 + 1:nrow(res)
```

For evaluation, we load our saved results and calculate rejection rates (we use an alpha of 0.10 since we are doing one-sided testing):

```
res = readRDS( file="data/discipline_simulation.rds" )

sres <- res %>% group_by( n_c, n_t, tx, model ) %>%
    summarise( E_est = mean( estimate ),
               SE = sd( estimate ),
               E_SE_hat = mean( std.error ),
               pow = mean( p.value <= 0.10 ) ) # one-sided testing
sres
```

```
## # A tibble: 378 x 8
## # Groups:   n_c, n_t, tx [54]
##      n_c   n_t    tx model   E_est     SE E_SE_hat
##    <dbl> <dbl> <dbl> <chr>   <dbl>  <dbl>    <dbl>
##  1     4     4  0.5  FE    -0.693  0.313    0.277
##  2     4     4  0.5  log   -0.694  0.476    0.430
##  3     4     4  0.5  log_~ -0.692  0.431    0.383
##  4     4     4  0.5  ratio -0.374  0.219    0.203
##  5     4     4  0.5  rati~ -0.291  0.151    0.139
##  6     4     4  0.5  raw   -0.719  0.535    0.515
##  7     4     4  0.5  simp~ -0.0195 0.0194   0.0157
##  8     4     4  0.75 FE    -0.292  0.310    0.274
##  9     4     4  0.75 log   -0.295  0.488    0.435
## 10     4     4  0.75 log_~ -0.305  0.424    0.373
## # i 368 more rows
## # i 1 more variable: pow <dbl>
```

19.4 Evaluating power

Once our simulation is run, we can explore power as a function of the design characteristics. In particular, we eventually want to calculate the chance of noticing effects of different sizes, given various sample sizes we might employ. Our driving question is how few schools on the treated side can we get away with? Also, we want to know how much having more schools on the control side allows us to get away with fewer schools on the treated side.

19.4.1 Checking validity of our models

Before we look at power, we need to check on whether our different models are valid. This is especiallt important as we are in a small n context, so we know asymptotics may not hold as they should. To check our models for validity we subset our trials to where `tx = 1`, and look at the rejection rates.

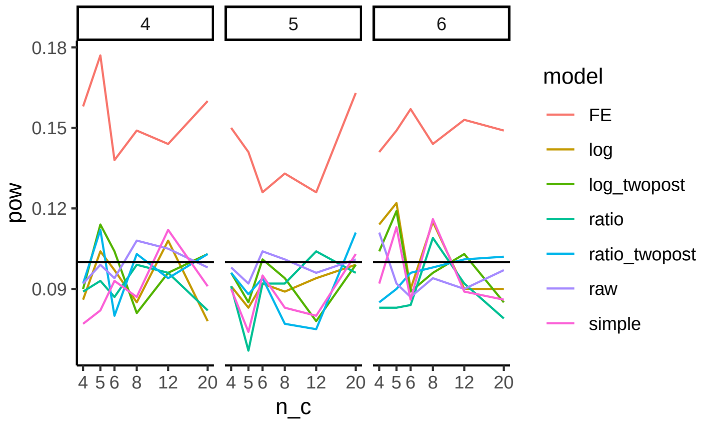
We first run a regression to see if rejection is a function of sample size (are smaller samples more invalid) and treatment-control imbalance. We center both variables so our intercepts are overall average rejection rates for each model considered:

```r
sres = mutate( sres,
               n = n_c + n_t,
               imbalance = pmax( n_t / n_c, n_c / n_t ) - 1 )
sres$n = (sres$n - mean(sres$n)) / sd(sres$n)
mod = lm( pow ~ 0 + (n + imbalance) * model - n - imbalance,
          data = filter( sres, tx == 1 ) )
broom::tidy(mod) %>%
  knitr::kable( digits = 3)
```

term	estimate	std.error	statistic	p.value
modelFE	0.143	0.006	24.593	0.000
modellog	0.099	0.006	16.999	0.000
modellog_twopost	0.093	0.006	15.939	0.000
modelratio	0.090	0.006	15.437	0.000
modelratio_twopost	0.093	0.006	15.967	0.000
modelraw	0.092	0.006	15.751	0.000
modelsimple	0.091	0.006	15.571	0.000
n:modelFE	-0.003	0.006	-0.459	0.647
n:modellog	0.001	0.006	0.141	0.888
n:modellog_twopost	-0.005	0.006	-0.919	0.360
n:modelratio	0.000	0.006	0.071	0.944
n:modelratio_twopost	0.002	0.006	0.414	0.680
n:modelraw	-0.006	0.006	-1.008	0.316
n:modelsimple	0.001	0.006	0.191	0.849
imbalance:modelFE	0.005	0.005	0.857	0.394
imbalance:modellog	-0.003	0.005	-0.587	0.558
imbalance:modellog_twopost	0.004	0.005	0.708	0.480
imbalance:modelratio	0.000	0.005	-0.001	0.999
imbalance:modelratio_twopost	0.001	0.005	0.249	0.804
imbalance:modelraw	0.006	0.005	1.154	0.251
imbalance:modelsimple	0.001	0.005	0.180	0.858

We can also plot the nominal rejection rates under the null:

```
sres %>% filter( tx == 1 ) %>%
ggplot( aes( n_c, pow, col=model ) ) +
  facet_wrap( ~ n_t, nrow=1 ) +
  geom_line() +
  geom_hline( yintercept = 0.10 ) +
  scale_x_log10(breaks=unique(sres$n_c) )
```



We see the fixed effect models have elevated rates of rejection. Interestingly, these rates do not seem particularly dependent on sample size or treatment-control imbalance (note lack of significant coefficeints on our regression model). The other models all appear valid.

We can also check for bias of our methods:

```
sres %>% group_by( model, tx ) %>%
  summarise( E_est = mean( E_est ) ) %>%
  pivot_wider( names_from="tx", values_from="E_est" )
```

```
## # A tibble: 7 x 4
## # Groups:   model [7]
##   model            `0.5`  `0.75`        `1`
##   <chr>            <dbl>   <dbl>      <dbl>
## 1 FE             -0.692  -0.290  -0.000703
## 2 log            -0.692  -0.288   0.00120
## 3 log_twopost    -0.692  -0.291   0.00241
## 4 ratio          -0.372  -0.187  -0.000937
## 5 ratio_twopost -0.289  -0.145  -0.00108
## 6 raw            -0.694  -0.290   0.00327
## 7 simple         -0.0206 -0.0104  0.0000998
```

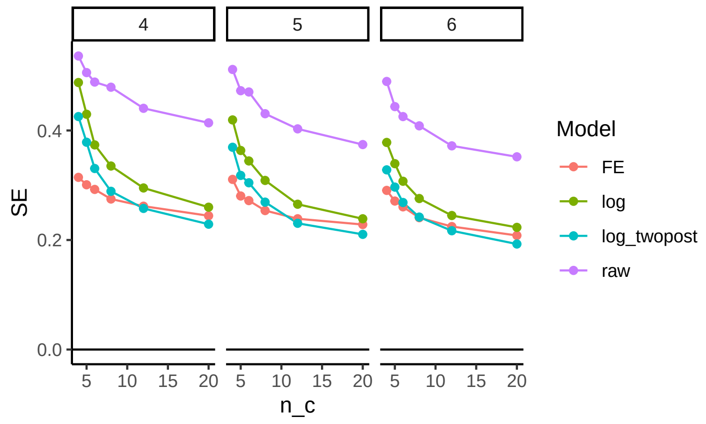
We see our models are estimating different things, none of which are the treatment effect as we parameterized it. In particular, “FE,” “log,” “raw,” and “log_twopost” are all estimating the impact on the log scale. Note that

$log(0.5) \approx -0.69$ and $log(0.75) \approx -0.29$. Our "simple" estimator is estimating the impact on the absolute scale; reducing discipline rates by 50% corresponds to about a 2% reduction in actual cases. Finally, "ratio" and "ratio_twopost" are estimating the change in the average ratio of post-policy discipline to pre; they are akin to a gain score as compared to the log regressions.

19.4.2 Assessing Precision (SE)

Now, which methods are the most precise? We look at the true standard errors across our methods (we drop "simple" and the "ratio" estimators since they are not on the ratio scale):

```
sres %>%
  group_by( model, n_c, n_t ) %>%
  summarise( SE = mean(SE ) ) %>%
  filter( !(model %in% c( "simple", "ratio", "ratio_twopost" ) ) ) %>%
  ggplot( aes( n_c, SE, col=model )) +
    facet_grid( . ~ n_t ) +
    geom_line() + geom_point() +
    geom_hline( yintercept = 0 ) +
  labs( colour = "Model" )
```

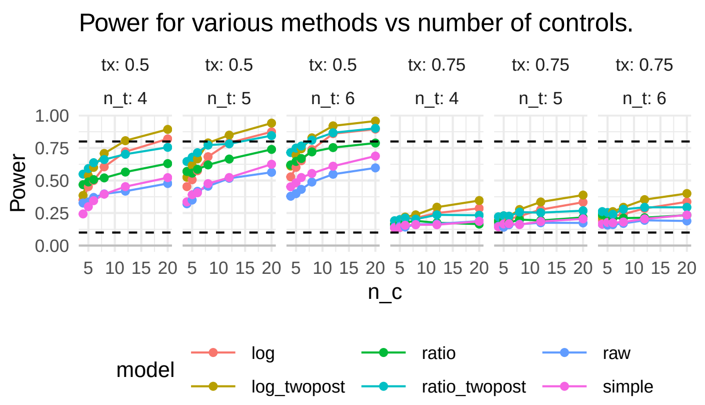


It looks like averaging two years for the outcome is helpful, and bumps up precision. The two way fixed effects model seems to react to the number of control units differently than the other estimators; it is way more precise when the number of controls is few, but the other estimators catch up. The "raw" estimator gives a baseline of no covariate adjustment; everything is substantially more precise than it. The covariates matter a lot.

19.4.3 Assessing power

We next look at power over our explored contexts, for the models that we find to be valid (i.e., not FE).

```r
sres %>%
  filter( model != "FE",tx != 1 ) %>%
  ggplot( aes( n_c, pow, col=model )) +
    facet_grid(  . ~ tx + n_t, labeller = label_both ) +
    geom_line() + geom_point() +
    geom_hline( yintercept = 0, col="grey" ) +
    geom_hline( yintercept = c( 0.10, 0.80 ), lty=2 ) +
  theme_minimal()+ theme( legend.position="bottom",
                          legend.direction="horizontal",
                          legend.key.width=unit(1,"cm"),
                          panel.border = element_blank() ) +
  labs( title="Power for various methods vs number of controls.",
      y = "Power" )
```



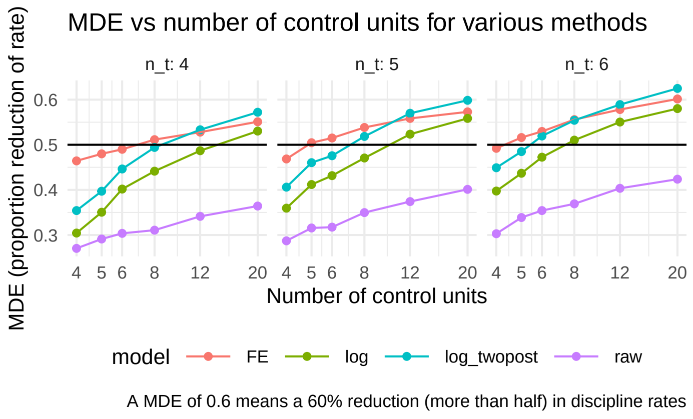
We mark 80% power with a dashed line. For a 25% reduction in discipline, nothing reaches desired levels of power. For 50% reduction, some designs do, but we need substantial numbers of control schools. Averaging two years of outcomes post-treatment seems important: the “twopost” methods have a distinct power bump. For a single year of outcome data, the log model seems our best bet.

19.4.4 Assessing Minimum Detectable Effects

Sometimes we want to know, given a design, what size effect we might be able to detect. The usual measure for this is the Minimum Detectable Effect (MDE), which is usually the size of the smallest effect we could detect with power 80%.

To calculate Minimal Detectable Effects (MDEs) for the log-scale estimators, we first average our SEs over our different designs, grouped by sample size, and then convert the SEs to MDEs by multiplying by 2.8. We then have to convert to our treatment scale by flipping the sign and exponentiating, to get out of the log scale.

```
sres2 = sres %>%
  group_by( model, n_c, n_t ) %>%
  summarise( SE = mean( SE ),
             E_SE_hat = mean( E_SE_hat ) ) %>%
  mutate( MDE = exp( - (1.64 + 0.8) * SE ) )

sres2 %>%
  filter( !(model %in% c( "simple", "ratio", "ratio_twopost" ) ) ) %>%
  ggplot( aes( n_c, MDE, col=model ) ) +
  facet_wrap( ~ n_t, labeller = label_both ) +
  geom_point() + geom_line()  +
  geom_hline( yintercept = 0.5 ) +
  theme_minimal() +
  scale_x_log10( breaks = unique( sres$n_c ) ) +
  theme( legend.position="bottom",
         legend.direction="horizontal", legend.key.width=unit(1,"cm"),
         panel.border = element_blank() ) +
  labs( x = "Number of control units", y = "MDE (proportion reduction of rate)",
        caption = "A MDE of 0.6 means a 60% reduction (more than half) in discipline ra
        title = "MDE vs number of control units for various methods" )
```



Corresponding with our findings regarding precision, above, the twopost estimator is the most sensitive, finding the smallest effects.

Chapter 20

Simulation under the Potential Outcomes Framework

If we are in the business of evaluating how various methods such as matching or propensity score weighting work in practice, we would probably turn to the potential outcomes framework for our simulations. The potential outcomes framework is a framework typically used in the causal inference literature to make very explicit statements regarding the mechanics of causality and the associated estimands one might target when estimating causal effects. While we recommend reading, for a more thourough overview, either [CITE Raudenbush or Field Experiments textbook], we briefly outline this framework here to set out our notation.

Take a sample of experimental units, indexed by i. For each unit, we can treat it or not. Denote treatment as $Z_i = 1$ for treated or $Z_i = 0$ for not treated. Now we imagine each unit has two potential outcomes being the outcome we would see if we treated it ($Y_i(1)$) or if we did not ($Y_i(0)$). Finally, our observed outcome is then

$$Y_i^{obs} = Z_i Y_i(1) + (1 - Z_i) Y_i(0).$$

For a unit, the treatment effect is $\tau_i = Y_i(1) - Y_i(0)$; it is how much our outcome changes if we treat vs. not treat. Frustratingly, for each unit we can only see one of its two potential outcomes, so we can never get an estimate of these individual τ_i. Under this view, causality is a missing data problem: if we only were able to impute the missing potential outcomes, we could have a dataset where we could calculate any estimands we wanted. E.g., the true average treatment effect *for the sample $\mathcal{S}$* would be:

$$ATE_{\mathcal{S}} = \frac{1}{N} \sum_i Y_i(1) - Y_i(0).$$

The average proportion increase, by contrast, would be

$$API_{\mathcal{S}} = \frac{1}{N} \sum_i \frac{Y_i(1)}{Y_i(0)}$$

20.1 Finite vs. Superpopulation inference

Consider a sample of n units, $\mathcal{S}$, along with their set of potential outcomes. We can talk about the true ATE of the sample, or, if we thought of the sample as being drawn from some larger population, we could talk about the true ATE of that larger population.

This is a tension that often arises in potential outcomes based simulations: if we are focused on $ATE_{\mathcal{S}}$ then for each sample we generate, our estimand could be (maybe only slightly) different, depending on whether our sample has more or fewer units with high τ_i. If, on the other hand, we are focused on where the units came from (which is our data generating model), our estimand is a property of the DGP, and would be the same for each sample generated.

The catch is when we calculate our performance metrics, we now have two possible targets to pick from. Furthermore, if we are targeting the superpopulation ATE, then our error in estimation may be due in part to the representativeness of the sample, *not* the estimation or uncertainty due to the random assignment.

We will follow this theme throughout this chapter.

20.2 Data generation processes for potential outcomes

If we want to write a simulation using the potential outcomes framework, it is clear and transparent to first generate a complete set of potential outcomes, then generate a random assignment based on some assignment mechanism, and finally generate the observed outcomes as a function of assignment and original potential outcomes.

For example, we might say that our data generation process is as follows: First generate each unit $i = 1, \ldots, n$, as

$$\begin{aligned} X_i &\sim exp(1) - 1 \\ Y_i(0) &= \beta_0 + \beta_1 X_i + \epsilon_i \text{ with } \epsilon_i \sim N(0, \sigma^2) \\ \tau_i &= \tau_0 + \tau_1 X_i + \alpha u_i \text{ with } u_i \sim t_{df} \\ Y_i(1) &= Y_i(0) + \tau_i \end{aligned}$$

with $exp(1)$ being the standard exponential distribution and t_{df} being a t distribution with df degrees of freedom. We subtract 1 from X_i to zero-center it (it is often convenient to have zero-centered covariates so we can then, e.g., interpret τ_0 as the true superpopulation ATE of our experiment).

The above model is saying that we first, for each unit, generate a covariate. We then generate our two potential outcomes. I.e., we are generating what the outcome would be for each unit if it were treated and if it were not treated. We are driving both the level and the treatment effect with X_i, assuming β_1 and τ_1 are non-zero.

One advantage of generating all the potential outcomes is we can then calculate the finite-sample estimands such as the true average treatment effect for the generated sample: we just take the average of $Y_i(1) - Y_i(0)$ for our sample.

Here is some code to illustrate the first part of the data generating process (we leave treatment assignment to later):

```r
gen_data <- function( n = 100,
                      R2 = 0.5,
                      beta_0 = 0, beta_1 = 1,
                      tau_0 = 1, tau_1 = 1,
                      alpha = 1, df = 3 ) {
  stopifnot( R2 >= 0 && R2 < 1 )
  X_i = rexp( n, rate = 1 ) - 1
  beta_1 = sqrt( 1 - R2 )
  sigma_e = sqrt( R2 )
  Y0_i = beta_0 + beta_1 * X_i + rnorm( n, sd=sigma_e )
  tau_i = tau_0 + tau_1 * X_i + alpha * rt( n, df = df )
  Y1_i = Y0_i + tau_i

  tibble( X = X_i, Y0 = Y0_i, Y1 = Y1_i )
}
```

And now we see our estimand can change:

```r
set.seed( 40454 )
d1 <- gen_data( 50 )
mean( d1$Y1 - d1$Y0 )
```

```
## [1] 0.6374925
```

```r
d2 <- gen_data( 50 )
mean( d2$Y1 - d2$Y0 )
```

```
## [1] 0.5479788
```

In reviewing our code, we know our superpopulation ATE should be `tau`, or 1 exactly. If our estimate for `d1` is 0.6 do we say that is close or far from the target? From a finite sample performance approach, we nailed it. From superpopulation, less so.

Also in looking at the above, there are a few details to call out:

- We can store the latent, intermediate quantities (both potential outcomes, in particular) so we can calculate the estimands of interest or learn about our data generating process. When we hand the data to an estimator, we would not provide this "secret" information.
- We are using a trick to index our DGP by an R2 value rather than coefficients on X so we can have a standardized control-side outcome (the expected variation of $Y_i(0)$ will be 1). The treatment outcomes will have more variation due to the heterogeneity of the treatment impacts.
- If we were generating data with a constant treatment impact, then $ATE_{\mathcal{S}} = ATE$ always; this is typical for many similations in the literature. That being said, treatment variation is what causes a lot of methods to fail and so having simulations that have this variation is usually important.

Once we have our *schedule of potential outcomes*, we would then generate the *observed outcomes* by assigning our (synthetic, randomly generated) n units to treatment or control. For example, say we wanted to simulate an observational context where treatment was a function of our covariate. We could model each unit as flipping a weighted coin with some probability that was a function of X_i as so:

$$
\begin{aligned}
p_i &= logit^{-1}(\xi_0 + \xi_1 X_i) \\
Z_i &= Bern(p_i) \\
Y_i &= Z_i Y_i(1) + (1 - Z_i) Y_i(0)
\end{aligned}
$$

Here is code for assigning our data to treatment and control:

```r
assign_data <- function( dat,
                         xi_0 = -1, xi_1 = 1 ) {
  n = nrow(dat)
  dat = mutate( dat,
                p = arm::invlogit( xi_0 + xi_1 * X ),
                Z = rbinom( n, 1, prob=p ),
                Yobs = ifelse( Z == 1, Y1, Y0 ) )
  dat
}
```

We can then add our assignment variable to our given data as so:

```
assign_data( d2 )
```

```
## # A tibble: 50 x 6
##           X     Y0      Y1     p     Z    Yobs
##       <dbl>  <dbl>   <dbl> <dbl> <int>   <dbl>
##  1  0.670    0.667   2.58  0.418     1   2.58
##  2  0.371    0.314   4.57  0.348     1   4.57
##  3  1.94     1.29    3.03  0.719     0   1.29
##  4 -0.244    0.119 -10.0   0.224     1 -10.0
##  5  0.00850  1.44    2.88  0.271     0   1.44
##  6  1.41     1.14    5.02  0.600     1   5.02
##  7 -0.864    0.461   0.802 0.134     1   0.802
##  8 -0.00533 -0.914  -1.17  0.268     0  -0.914
##  9 -0.907   -0.202   0.555 0.129     1   0.555
## 10 -0.363   -0.141   1.16  0.204     1   1.16
## # i 40 more rows
```

Note how `Yobs` is, depending on `Z`, either `Y0` or `Y1`. Separating our our DGP and our random assignment underscores the potential outcomes framework adage of the data are what they are, and we the experimenters (or nature) is randomly assigning these whole units to various conditions and observing the consequences.

In general, we might instead put the `p_i` part of the model in our code generating the outcomes, if we wanted to view the chance of treatment assignment as inherent to the unit (which is what we usually expect in an observational context).

20.3 Finite sample performance measures

Let's generate a single dataset with our DGP from above, and run a small experiment where we actually randomize units to treatment and control:

```
n = 100
set.seed(442423)
dat = gen_data(n, tau_1 = -1)
dat = mutate( dat,
              Z = 0 + (sample( n ) <= n/2),
              Yobs = ifelse( Z == 1, Y1, Y0 ) )
mod = lm( Yobs ~ Z, data=dat )
coef(mod)[["Z"]]
```

```
## [1] 0.8914992
```

We can compare this to the true finite-sample ATE:

```
mean( dat$Y1 - dat$Y0 )
```

```
## [1] 1.154018
```

Our finite-population simulation would be:

```
rps <- rerun( 1000, {
  dat = mutate( dat,
                Z = 0 + (sample( n ) <= n/2),
                Yobs = ifelse( Z == 1, Y1, Y0 ) )
  mod = lm( Yobs ~ Z, data=dat )
  tibble( ATE_hat = coef(mod)[["Z"]],
          SE_hat = arm::se.coef(mod)[["Z"]] )
  }) %>%
  bind_rows()

rps %>% summarise( EATE_hat = mean( ATE_hat ),
                   SE = sd( ATE_hat ),
                   ESE_hat = mean( SE_hat ) )
```

```
## # A tibble: 1 x 3
##   EATE_hat    SE ESE_hat
##      <dbl> <dbl>   <dbl>
## 1     1.16 0.248   0.307
```

We are simulating on a single dataset. In particular, our set of potential outcomes is entirely fixed; the only source of randomness (and thus the randomness behind our SE) is the random assignment. Now this opens up some room for critique: what if our single dataset is non-standard?

Our super-population simulation would be, by contrast:

```
rps_sup <- rerun( 1000, {
  dat = gen_data(n)
  dat = mutate( dat,
                Z = 0 + (sample( n ) <= n/2),
                Yobs = ifelse( Z == 1, Y1, Y0 ) )
  mod = lm( Yobs ~ Z, data=dat )
  tibble( ATE_hat = coef(mod)[["Z"]],
          SE_hat = arm::se.coef(mod)[["Z"]] )
  }) %>%
  bind_rows()

rps_sup %>% summarise( EATE_hat = mean( ATE_hat ),
                   SE = sd( ATE_hat ),
                   ESE_hat = mean( SE_hat ))
```

```
## # A tibble: 1 x 3
##   EATE_hat    SE ESE_hat
##      <dbl> <dbl>   <dbl>
## 1     1.00 0.381   0.378
```

First, note our superpopulation simulation is not biased for the superpopulation ATE. Also note the true SE is larger than our finite-sample simulation; this is because part of the uncertainty in our estimator is the uncertainty of whether our sample is representative of the superpopulation.

Finally, this clarifies that our linear regression estimator is estimating standard errors assuming a superpopulation model. The true finite sample standard error is less than the expected estimated error: from a finite sample perspective, our estimator is giving overly conservative uncertainty estimates. (This discrepancy is often called the correlation of potential outcomes problem.)

20.4 Nested finite simulation procedure

We just saw a difference between a specific, single, finite-sample dataset and a superpopulation. What if we wanted to know if this phenomenon was more general across a set of datasets? This question can be levied more broadly: if we run a simulation on a single dataset, this is even more narrow than running on a single scenario: if we compare methods and find one is superior to another for our single dataset, how do we know this is not an artifact of some specific characteristic of *that data* and not a general phenomonen at all?

One way forward is to run a nested simulation, where we generate a series of finite sample datasets, and then for each dataset run a small simulation. We then calculate the expected finite sample performance across the datasets. One could almost think of the datasets themselves as a "factor" in our multifactor experiment. This is what we did in [CITE estimands paper]

Borrowing from the simulation appendix of [CITE estimands paper], repeat R times:

1. Generate a dataset using a particular DGP. This data generation is the "sampling step" for a superpopulation (SP) framework. The DGP represents an infinite superpopulation. Each dataset includes, for each observation, the potential outcome under treatment or control.
2. Record the true finite-sample ATE, both person and site weighted.
3. Then, three times, do a finite simulation as follows:

a. Randomize units to treatment and control.
b. Calculate the corresponding observed outcomes.
c. Analyze the results using the methods of interest, recording both the point estimate and estimated standard error for each.

Having only three trials will give a poor estimate of within-dataset variability for each dataset, but the average across the R datasets in a given scenario gives a reasonable estimate of expected variability across datasets of the type we would see given the scenario parameters.

To demonstrate we first make a mini-finite sample driver:

```
one_finite_run <- function( R0 = 3, n = 100, ... ) {
  dat = gen_data( n = n, ... )
  rps <- rerun( R0, {
         dat = mutate( dat,
                    Z = 0 + (sample( n ) <= n/2),
                    Yobs = ifelse( Z == 1, Y1, Y0 ) )
        mod = lm( Yobs ~ Z, data=dat )
        tibble( ATE_hat = coef(mod)[["Z"]],
                SE_hat = arm::se.coef(mod)[["Z"]] )
    }) %>%
    bind_rows()
  rps$ATE = mean( dat$Y1 - dat$Y0 )
  rps
}
```

This driver also stores the finite sample ATE for future reference:

```
one_finite_run()
```

```
## Warning: `rerun()` was deprecated in purrr 1.0.0.
## i Please use `map()` instead.
##   # Previously
## rerun(3, {
## dat = mutate(dat, Z = 0 + (sample(n) <= n / 2),
## Yobs = ifelse(Z == 1, Y1, Y0))
## mod = lm(Yobs ~ Z, data = dat)
## tibble(ATE_hat = coef(mod)[["Z"]], SE_hat =
## arm::se.coef(
## mod)[["Z"]])
## })
##
##   # Now
## map(1:3, ~ {
## dat = mutate(dat, Z = 0 + (sample(n) <= n / 2),
## Yobs = ifelse(Z == 1, Y1, Y0))
## mod = lm(Yobs ~ Z, data = dat)
## tibble(ATE_hat = coef(mod)[["Z"]], SE_hat =
## arm::se.coef(
## mod)[["Z"]])
## })
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to
## see where this warning was generated.
```

```
## # A tibble: 3 x 3
##   ATE_hat SE_hat   ATE
##     <dbl>  <dbl> <dbl>
## 1   0.348  0.421 0.768
## 2   1.32   0.472 0.768
## 3   1.17   0.549 0.768
```

We then run a bunch of finite runs.

```
runs <- rerun( 500, one_finite_run() ) %>%
  bind_rows( .id = "runID" )
```

```
## Warning: `rerun()` was deprecated in purrr 1.0.0.
## i Please use `map()` instead.
##   # Previously
##   rerun(500, one_finite_run())
##
##   # Now
##   map(1:500, ~ one_finite_run())
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to
## see where this warning was generated.
```

We use `.id` because we will need to separate out each finite run and analyze separately, and then aggregate.

Each finite run is a very noisy simulation for a fixed dataset. This means when we calculate performance measures we have to be careful to avoid bias in the calculations; in particular, we need to focus on estimating SE^2 across the finite runs, not SE, to avoid the bias caused by having a few observations with every estimate.

```
fruns <- runs %>% group_by( runID ) %>%
  summarise( EATE_hat = mean( ATE_hat ),
             SE2 = var( ATE_hat ),
             ESE_hat = mean( SE_hat ),
             .groups = "drop" )
```

And then we aggregate our finite sample runs:

```
res <- fruns %>%
  summarise( EEATE_hat = mean( EATE_hat ),
             EESE_hat = sqrt( mean( ESE_hat^2 ) ),
             ESE = sqrt( mean( SE2 ) ) ) %>%
  mutate( calib = 100 * EESE_hat / ESE )
```

```
res
```

```
## # A tibble: 1 x 4
##   EEATE_hat EESE_hat   ESE calib
##       <dbl>    <dbl> <dbl> <dbl>
## 1     0.996    0.380 0.331  115.
```

We see our expected standard error estimate is, across the collection of finite sample scenarios all sharing a similar parent superpopulation DGP, 15% too large for the true expected finite-sample standard error.

We need to keep the squaring. If we look at the SEs themselves, we have further apparent bias due to our *estimated* `ESE_hat` being so unstable due to too few observations:

```
mean( sqrt( fruns$SE2 ) )
```

```
## [1] 0.2944556
```

We can use our collection of mini-finite-sample runs to estimate superpopulation quantities as well. Given that the simulation datasets are i.i.d. draws, we can simply take expectations across all our simulations. The only concern is our estimates of MCSE will be off due to the clustering in our simulation runs.

Here we calculate superpopulation performance measures (both with the squared SE and without; we prefer the squared version):

```
runs %>%
  summarise( EATE_hat = mean( ATE_hat ),
             SE_true = sd( ATE_hat ),
             SE_hat = mean( SE_hat ),
             SE2_true = var( ATE_hat ),
             SE2_hat = mean( SE_hat^2 ) ) %>%
  pivot_longer( cols = c(SE_true:SE2_hat ),
                names_to = c( "estimand", ".value" ),
                names_sep ="_" ) %>%
  mutate( inflate = 100 * hat / true )
```

```
## # A tibble: 2 x 5
##   EATE_hat estimand  true   hat inflate
##      <dbl> <chr>    <dbl> <dbl>   <dbl>
## 1    0.996 SE       0.389 0.377    96.9
## 2    0.996 SE2      0.151 0.142    93.9
```

Chapter 21

Simulations as evidence

We began this book with an acknowledgement that simulation is fraught with the potential for misuse: *simulations are doomed to succeed.* We close by reiterating this point, and also disussing several ways researchers might argue for their simulations being more useful than typical.

In particular a researcher might do any of the following

1. Use extensive multi-factor simulations
2. Beat them at their own game. Generate simulations based on prior literature.
3. Build calibrated simulations

21.1 Use extensive multi-factor simulations

"If a single simulation is not convincing, use more of them," is one principle a reseacher might take. By conducting extensive multifactor simulations, once can explore a large space of possible data generating scenarios. If, across the full range of scenarios, a general story bears out, then perhaps that will be more convincing than a narrower range.

Of course, the critic will claim that some aspect that is not varying is the real culprit. If this is unrealistic, then the findings, across the board, may be less relevant. Thus, pick the factors one varies with care.

21.2 Beat them at their own game

If a prior paper uses a simulation to make a case, one approach is to replicate that simulation, adding in the new estimator one wants to evaluate. This makes it (more) clear that you are not fishing: you are using something established in the literature as a published benchmark. By constraining oneself to published simulations, one has less wiggle room to cherry pick a data generating process that works the way you want.

21.3 Calibrated simulations

A practice in increasing vogue is to generate *calibrated simulations*. These are simulations tailored to a specific applied contexts, with the results of the simulation studies designed to more narrowly inform what assumptions and structures are necessary in order to make progress in that context.

Often these simulations are built out of existing data. For example, one might sample, with replacement, from the covariate distribution of an actual dataset so that the distribution of covariates is authentic in how the covariates are distributed and, more importantly, how they co-relate.

The problem with this approach is one still needs to generate a ground truth to assess how well the estimators work in practice. It is very easy to accidently put a very simple model in place for this component, thus making a calibrated simulation quite naive in the very way that counts.

The potential outcomes framework provides a natural way of generating calibrated simulations (Kern et al., 2014). Calbirated simulations are simulations tailored to specific real-world scenarios, to maximize their face validity as being representative of something we would see in practice. One way to generate a calibrated simulation is to use an existing datasets to generate plausible scenarios.

One way to do this with potential outcomes is to take an existing randomized experiment or observational study and impute all the missing potential outcomes under some specific scheme. This fully defines the sample of interest and thus any target parameters, such as a measure of heterogeneity, are then known. We will then synthetically, and repeatedly, randomize and "observe'' outcomes to be analyzed with the methods we are testing. We could also resample from our dataset to generate datasets of different size, or to have a superpopulation target as our estimand.

The key feature is the imputation step. One baseline method one can use is to generate a matched-pairs dataset by, for each unit, finding a close match given all the demographic and other covariate information of the sample. We then use the matched unit as the imputed potential outcome.
By doing this (with replacement) for all units we can generate a fully imputed dataset which we then use as our population. This can preserve complex relationships in the data that are not model dependent. In particular, if outcomes tend to be coarsely defined (e.g., on an integer scale) or have specific clumps

(such as zero-inflation or rounding), this structure will be preserved.

One concern with this approach is the noise in the matching will in general dilute the structure of the treatment effect.
This is akin to measurement error diluting found relationships in linear models. We can then sharpen these relationships towards a given model by first imputing missing outcomes using a specified model, and then matching on all units including the imputed potential outcome. This is not a data analysis strategy, but instead a method of generating synthetic data that both has a given structure of interest and also remains faithful to the idiosyncrasies of an actual dataset.

A second approach that allows for varying the level of a systematic effect is to specify the treatment effect model and impute treatment outcomes for all control units.
Then the complex structure between covariates and $Y(0)$ would be perfectly preserved. Unfortunately, this would give 0 idiosyncratic treatment variation (unit-to-unit variation in the τ_i that is not explained by a model). To add in idiosyncratic variation we would then need to generate a distribution of perturbations and add these to the imputed outcomes just as an error term in a regression model.

Regardless, once a fully observed sample has been obtained we can investigate several aspects of our estimators as above.

Chapter 22

The Parametric bootstrap

An inference procedure very much connected to simulation studies is the parametric bootstrap. The parametric bootstrap is a bootstrap technique designed to obtain standard error estimates for an estimated parametric model. It can do better than the case-wise bootstrap in some circumstances, usually when there is need to avoid the discrete, chunky nature of a casewise bootstrap (which will only give values that exist in the original dataset).

For a parametric bootstrap, the core idea is to fit a given model to actual data, and then take the parameters we estimate from that model as the DGP parameters in a simulation study. The parametric bootstrap is a simulation study for a specific scenario, and our goal is to assess how variable (and, possibly, biased) our estimator is for this specific scenario. If the behavior of our estimator in our simulated scenario is similar to what it would be under repeated trials in the real world, then our bootstrap answers will be informative as to how well our original estimator performs in practice. This is the bootstrap principle, or analogy with an additional assumption that the real-world is effectively well specified as the parameteric model we are fitting.

In particular we do the following:

1. generate data from a model with coefficients as estimated on the original data.
2. repeatedly estimate our target quantity on a series of synthetic data sets, all generated from this model.
3. examine this collection of estimates to assess the character of the estimates themselves, i.e. how much they vary, whether we are systematically estimating too high or too low, and so forth.
4. The variance and bias of our estimates in our simulation is probably like the actual variance and bias of our original estimate (this is precisely the bootstrap analogy).

A key feature of the parametric bootstrap is it is not, generally, a multifactor simulation experiment. We fit our model to the data, and use our best estimate of the world, as given by the fit model, to generate our data. This means we generally want to simulate in contexts that are (mostly) *pivotal*, meaning the distribution of our test statistic or point estimate is relatively stable across different scenarios. In other words, we want the uncertainty of our estimator to not heavily depend on the exact parameter values we use in our simulation, so that if we are simulating with incorrect parameters our bootstrap analogy will still hold.

Often, to achieve a reasonable claim of being pivotal, we will focus on standardized statistics, such as the t-statistic of

$$t = \frac{est}{\widehat{SE}}$$

It is more common for the distribution of a standardized test statistic to have a canonical distribution across scenarios than an absolute estimate.

22.1 Air conditioners: a stolen case study

Following the case study presented in [CITE bootstrap book], consider some failure times of air conditioning units:

```
dat = c( 3, 5, 7, 18, 43, 85, 91, 98, 100, 130, 230, 487 )
```

We are interested in the log of the average failure time:

```
n = length(dat)
y.bar = mean(dat)
theta.hat = log( y.bar )

c( n = n, y.bar = y.bar, theta.hat = theta.hat )
```

```
##          n      y.bar  theta.hat
##  12.000000 108.083333   4.682903
```

We are interested in this because we are modeling the failure time of the air conditioners with an exponential distribution. This means we will generate new failure times with an exponential distribution:

```
reps = replicate( 10000, {
    smp = rexp(n, 1/y.bar)
    log( mean( smp ) )
})

res_par = tibble(
  bias.hat = mean( reps ) - theta.hat,
  var.hat = var( reps ),
  CIlog_low = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.975),
  CIlog_high = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.025),
  CI_low = exp( CIlog_low ),
  CI_high = exp( CIlog_high ) )
res_par
```

```
## # A tibble: 1 x 6
##   bias.hat var.hat CIlog_low CIlog_high CI_low
##      <dbl>   <dbl>     <dbl>      <dbl>  <dbl>
## 1  -0.0420  0.0856      4.07       5.21   58.4
## # i 1 more variable: CI_high <dbl>
```

Note how we are, as usual, in our standard simulation framework of repeatidly (1) generating data and (2) analyzing the simulated data. Nothing is changed.

The nonparametric, or case-wise, bootstrap (this is what people normally mean when they say bootstrap) would look like this:

```
reps = replicate( 10000, {
    smp = sample( dat, replace=TRUE )
    log( mean( smp ) )
})

res_np = tibble(
  bias.hat = mean( reps ) - theta.hat,
  var.hat = var( reps ),
  CIlog_low = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.975),
  CIlog_high = theta.hat + bias.hat - sqrt(var.hat) * qnorm(0.025),
  CI_low = exp( CIlog_low ),
  CI_high = exp( CIlog_high ) )


bind_rows( parametric = res_par,
           casewise = res_np, .id = "method") %>%
  mutate( length = CI_high - CI_low )
```

```
## # A tibble: 2 x 8
##   method     bias.hat var.hat CIlog_low CIlog_high
##   <chr>         <dbl>   <dbl>     <dbl>      <dbl>
## 1 parametric  -0.0420  0.0856      4.07       5.21
## 2 casewise    -0.0651  0.132       3.90       5.33
## # i 3 more variables: CI_low <dbl>,
## #   CI_high <dbl>, length <dbl>
```

This is *also* a simulation: our data generating process is a bit more vague, however, as we are just resampling the data. This means our estimands are not as clearly specified. For example, in our parameteric approach, our target parameter is known to be true. In the case-wise, the connection between our DGP and the parameter `theta.hat` is less explicit.

Overall, in this case, our parametric bootstrap can model the tail behavior of an exponential better than case-wise. Especially considering the small number of observations, it is going to be a more faithful representation of what we are doing–provided our model is well specified for the real world distribution.

Chapter 23

Further readings and resources

We close with a list of things of interest we have discovered while writing this text.

- Morris, White, & Crowther (2019). Using simulation studies to evaluate statistical methods.
- High-level simulation design considerations.
- Details about performance criteria calculations.
- Stata-centric.
- SimDesign R package (Chalmers, 2019)
- Tools for building generic simulation workflows.
- Chalmers & Adkin (2019). Writing effective and reliable Monte Carlo simulations with the SimDesign package.
- DeclareDesign (Blair, Cooper, Coppock, & Humphreys)
- Specialized suite of R packages for simulating research designs.
- Design philosophy is very similar to "tidy" simulation approach.
- SimHelpers R package (Joshi & Pustejovsky, 2020)
- Helper functions for calculating performance criteria.
- Includes Monte Carlo standard errors.

Bibliography

Abdulkadiroğlu, A., Angrist, J. D., Narita, Y., and Pathak, P. A. (2017). Research design meets market design: Using centralized assignment for impact evaluation. *Econometrica*, 85(5):1373–1432.

Gerber, A. S. and Green, D. P. (2012). *Field Experiments: Design, Analysis, and Interpretation.* W. W. Norton & Company. W. W. Norton & Company.

Kern, H. L., Stuart, E. A., Hill, J., and Green, D. P. (2014). Assessing Methods for Generalizing Experimental Impact Estimates to Target Populations. *Journal of Research on Educational Effectiveness*, 9(1):103–127.

Miratrix, L. W., Weiss, M. J., and Henderson, B. (2021). An applied researcher's guide to estimating effects from multisite individually randomized trials: Estimands, estimators, and estimates. *Journal of Research on Educational Effectiveness*, 14(1):270–308.

Staiger, D. O. and Rockoff, J. E. (2010). Searching for effective teachers with imperfect information. *Journal of Economic perspectives*, 24(3):97–118.

Sundberg, R. (2003). Conditional statistical inference and quantification of relevance. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 65(1):299 – 315.